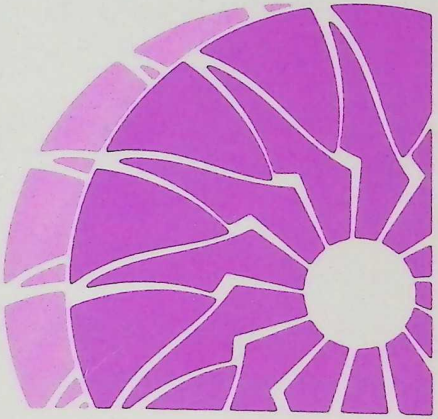
TURBO PASCAL

Volume 2
TSRs and More

Turbo Professional®

A library of over 600 essential procedures and functions, optimized for non-object-oriented programming in Turbo Pascal. Includes ■ Pop-up resident routines ■ Windows ■ Menus ■ Interrupt service routines ■ Data entry routines ■ Powerful and useful demo programs and more.



TURBO Power

Turbo Professional

User's Manual

Volume 2
TSRs and More

Copyright (c) 1988, 1993, TurboPower Software
All Rights Reserved

Second Edition July 1993

Order line (U.S. and Canada): 800-333-4160
Elsewhere: 719-260-9136

TurboPower Software
P.O. Box 49009
Colorado Springs, CO 80949-9009

Technical Support: 719-260-6641
Fax: 719-260-7151
9 a.m. to 5 p.m. Mountain Time, Monday-Friday

CompuServe 76004,2611
PCVENB Section 6

Turbo Professional is copyright (c) 1987, 1988, 1993 by TurboPower Software, all rights reserved. Portions are copyright (c) 1985, 1986 by Sunny Hill Software, and are used under license to TurboPower Software. All TurboPower Software products are trademarks or registered trademarks of TurboPower Software. Other brand and product names are trademarks or registered trademarks of their respective holders.

Table of Contents

Volume 1: Screen Management

1. Introduction **1**
- A. System Requirements 3
- B. Files Supplied 4
- C. Getting Started 6
- D. Purchase Agreement 17

2. Screen Handling **19**
- A. Overview 19
- B. TPCRT: Enhanced CRT Unit 21
- C. TPWINDOW: Popup Windows 83
- D. TPMENU: Pulldown Menus 114
- E. TPEDIT: Line Editor 153
- F. TPHELP: Popup Help 168
- G. TPMOUSE: Mouse Routines 199
- H. TPPICK: Pick Lists 231
- I. TPDIR: Popup Directory Picker 252
- J. TPSCREEN: Virtual Screens 260

3. Data Entry **285**
- A. Overview 285
- B. TPENTRY: Data Entry Screens 287
- C. TPDATE: Date and Time Variables 378
- D. TPMEMO: Memo Field Editor 402

Identifiers

Index

Volume 2: TSRs and More

4. String Handling 421
- A. Overview 421
- B. TPSTRING: String Manipulation 423
- C. TPASCIIZ: Long ASCIIZ Strings 465
- D. TPSTRDEV: Formatted Strings 484

5. DOS/BIOS Calls 487
- A. Overview 487
- B. TPDOS: Assorted DOS/BIOS Calls 489
- C. TPENHKBD: Enhanced Keyboard Interface 518

6. Interrupts and TSRs 525
- A. Overview 525
- B. TPINT: ISR Management 528
- C. TPTSR: TSR Management 542
- D. TP8087: 8087 Support for TSRs 566
- E. TPINT24: Critical Error Handler 575

7. Sorting 579
- A. Overview 579
- B. TPSORT: In-memory Sort Routine 582

8. Memory Allocation 587
- A. Overview 587
- B. TPEMS: EMS Memory Management 590
- C. TPEXTMEM: Extended Memory Management 602

9. Macros 609
- A. Overview 609
- B. TPMACRO: Keyboard Macros 611
- C. TPMACED: Macro Editor 626

10. Large Arrays 629
- A. Overview 629
- B. TPRARRAY: Arrays in Normal RAM 632
- C. TPEARRAY: Arrays in EMS Memory 644
- D. TPVARRAY: Arrays in Virtual Memory 653

11. BCD Arithmetic 665
A. Overview 665
B. TPBCD: High-precision Reals 667

12. Miscellaneous 687
A. Overview 687
B. TPERRHAN: Runtime Error Recovery 688
C. TPINLINE: Assorted Inline Macros 692

13. Demonstration Programs 705
A. SMACS: Macro Processor 706
B. PCALC: Programmer's Calculator 710
C. PTIME: Programmer's Time Management Utility 715
D. PTRPT: Report Generator for PTIME 718
E. PREF: Programmer's Quick Reference Chart 721
F. MAKEMENU: Menu Maker for TPMENU 724
G. BSORT: File Sorting Utility 735
H. TFIND: Text Find Utility 736
I. REP: Command Repeater 738
J. DIFF: Difference Finder 741
K. VBROWSE: Virtual File Browser 742
L. ENTRY: Data Entry Example 745
M. Other Programs 749

14. Appendices 751
A. Unit Dependencies 751
B. Keyboard Handling in Turbo Professional 753
C. Using Overlays with Turbo Professional 763

Identifiers

Index

4. String Handling

A. Overview

Almost every commercial library contains a variety of routines for string manipulation, and Turbo Professional is no exception. As expected, you'll find the usual routines for converting numbers to strings in a variety of formats, changing the case of alphabetic characters (including those in the international character set) in a string, trimming leading and trailing blanks, padding strings with blanks, and doing comparisons of strings. But you'll also find a significant number of useful routines that you probably didn't expect.

The TPSTRING unit consists primarily of general purpose string manipulation routines, including all of the categories of routines mentioned above. It also contains a number of slightly more unusual routines. Most notable is the complete set of routines for massaging DOS file and path names. These allow you to add and remove extensions and path identifiers, clean up filenames with extraneous characters, and turn valid file names into complete pathnames in Drive:\Path\Filename.Ext format. TPSTRING also includes routines to conveniently allocate and deallocate heap space for string variables; to search large arrays for the location of a specified string; and to convert strings to Soundex codes, allowing you to search for names that almost match, such as "Smith" and "Smythe".

One of the most flexible routines in the unit is Form, which is modelled on the Form function in the BCD version of Turbo Pascal 3.0. Form takes two parameters, a string Mask and a real number, and returns a formatted string that merges individual digits from the real into the appropriate locations in the Mask. For example, given the Mask "$##,###,###.##" and the real value 12300.26666, Form would return " $12,300.27". (Note that a second version of the Form function can be found in the TPBCD unit, one designed especially for high-precision BCD reals.)

One fairly common complaint about Turbo Pascal is that its strings are limited to 255 characters. Although 255 characters is usually more than adequate, when you really do need longer strings, you're stuck. The TPASCIIZ unit offers a comprehensive solution to this problem. Unlike normal Turbo strings, an ASCIIZ string contains no length byte. Instead, it marks the end of a string with a single null character (#0, hence the Z in ASCIIZ). The inherent disadvantage of ASCIIZ's is that certain operations, such as checking the length of a string, take more time, but the great advantage is that they can be much, much longer. (Four times as long by default, but up to 65,520 characters if desired.) Of course, the other potential disadvantage of using ASCIIZ strings is that the Turbo compiler does not support

them, but TPASCIIZ contains just about everything you might need to work with them. Besides providing equivalents for Turbo's own string manipulation routines--such as Concat, Insert, Delete, Pos, and Length--it also contains equivalents for most of the basic routines in TPSTRING, including routines for case conversion, string comparisons, blank trimming, blank padding, and dynamic allocation of ASCIIZ variables. Equally important, it contains routines for reading and writing ASCIIZ strings to and from text files.

The tiny TPSTRDEV unit serves a single function: it allows you to use the power of the Write statement to create formatted strings, without sending the output to a file. This ability is particularly useful if you are using the FastWrite routine in TPCRT, which won't accept a number as a parameter, and won't pad strings with blanks. For example, if you wanted to convert the integer values -357 and 41 into a single string 18 characters wide, you *could* use the Long2Str and Pad functions in TPSTRING, then send the cumulative result of the several function calls to FastWrite. If you were using the TPSTRDEV unit, however, you could simply say

```
Write(TpStr, -357, 41:14);
FastWrite(ReturnStr, Row, Column, Attribute);
```

More generally, TPSTRDEV allows you to use the formatting capabilities provided by Write statements in situations where you are not writing to the screen or to a text file.

B. TPSTRING: String Manipulation

As mentioned previously, TPSTRING contains all of the string manipulation routines that you would expect, and a number that you might not expect. We think it has just about everything a professional program will need for basic string manipulation. TPSTRING works with Turbo's standard 255 character strings. If you need something longer, check out the TPASCIIZ chapter.

The routines in TPSTRING fall into several categories: numeric conversion and formatting; general purpose manipulation such as case changes, padding, trimming and centering; string comparison and searching; dynamic storage of strings on the heap; and DOS pathname parsing.

Types

```
BTable = array[0..255] of Byte;
```

A table used by the Boyer-Moore search routines.

```
CharSet = set of Char;
```

A set of characters.

```
CompareType = (Less, Equal, Greater);
```

The result of a comparison of two variables.

```
{$IFOPT N+}
Float = Extended;
{$ELSE}
Float = Real;
{$ENDIF}
```

If TPSTRING is compiled with $N+, Float is defined as an Extended rather than a Real. The definition of Float affects the Real2Str and Str2Real routines.

Constants

```
CommaForPeriod : Boolean = False;
```

If True, the Form function will return '.' for ',' and vice versa. This makes it a bit easier to write programs that can be used both in the U.S. and in countries that use commas for decimal points.

```
LetterValues : array['A'..'Z'] of Byte =
(3 {A} , 6 {B} , 5 {C} , 4 {D} , 3 {E} , 5 {F} , 5 {G} , 4 {H} , 3 {I} ,
 8 {J} , 7 {K} , 4 {L} , 5 {M} , 3 {N} , 3 {O} , 5 {P} , 7 {Q} , 4 {R} ,
 3 {S} , 3 {T} , 4 {U} , 6 {V} , 5 {W} , 8 {X} , 8 {Y} , 9 {Z} );
```

A table used by CompareLetterSets for estimating word similarity.

```
MoneySign : Char = '$';
```

This character is used when displaying values in dollars-and-cents format. If desired, you could change this constant to #156, for example, allowing Form to display amounts in terms of British pounds rather than dollars.

AddBackSlash / BinaryB

Declaration

```
function AddBackSlash(DirName : string) : string;
```

Purpose

Add a default backslash to a directory name.

Comments

A backslash character '\' separates an MS-DOS directory name from a filename contained within that directory. A special case prevents the simple concatenation of a backslash to all directory names, however. The root directory name actually consists of a backslash, and DOS becomes unhappy if an additional backslash is appended. The AddBackSlash function concatenates a backslash to a directory name only when the directory name does not already end in a backslash.

Examples

```
PathName := AddBackSlash('C:\')+'AUTOEXEC.BAT';
```

The resulting PathName is 'C:\AUTOEXEC.BAT'

```
PathName := AddBackSlash('\turbo\test')+'BUG.PAS';
```

The resulting PathName is '\turbo\test\BUG.PAS'

See Also

FullPathName

Declaration

```
function BinaryB(B : Byte) : string;
```

Purpose

Return binary string for Byte.

Comments

Returns an 8 character string which represents the binary value of the byte. Leading zeros are included.

Examples

```
BinaryB(0)
```

Returns '00000000'

```
BinaryB($B1)
```

Returns '10110001'

See Also

BinaryL BinaryW

Declaration

```
function BinaryL(L : LongInt) : string;
```

Purpose

Return binary string for LongInt.

Comments

Returns a 32 character string which represents the binary value of a LongInt. No separation characters are inserted within the string. Leading zeros are included.

Examples

```
BinaryL(-1)
```

Returns '11111111111111111111111111111111'

```
BinaryL(7)
```

Returns '00000000000000000000000000000111'

See Also

BinaryB BinaryW

Declaration

```
function BinaryW(W : Word) : string;
```

Purpose

Return binary string for Word.

Comments

Returns a 16 character string which represents the binary value of a Word. No separation characters are inserted within the string. Leading zeros are included.

Examples

```
BinaryW($B800)
```

Returns '1011100000000000'

```
BinaryW(-1)
```

Incorrect! This won't compile because Words are always non-negative. Use BinaryL if negative numbers are needed.

See Also

BinaryB BinaryL

BMMakeTable

Declaration

```
procedure BMMakeTable(MatchString : string; var BT : BTable);
```

Purpose

Build Boyer-Moore link table.

Comments

The Boyer-Moore algorithm is a very fast string search method. It gains this speed from the fact that it need not compare many elements of the string. The elements it can skip are determined by a link table that must be initialized before the search itself begins.

The link table is an array[0..255] of byte. For each character in the ASCII set (0..255) the table contains the number of positions the search can safely skip. BMMakeTable takes the string to be matched, MatchString, and returns a properly initialized link table. The link table will be passed to the Boyer-Moore search routine itself to aid in the search.

The BMMakeTable operation may dominate the performance of B-M if the region to be searched is short. In general, you should minimize the number of calls to BMMakeTable. If you find the need to call it very often, you will probably get better performance using the brute force function Search.

See the BMSearch function for an example.

See Also

BMSearch

BMSearchUC

Declaration

```
function BMSearch(var Buffer; BufLength : Word;
                  BT : BTable; MatchString : string) : Word;
```

Purpose

Search Buffer for MatchString using the Boyer-Moore algorithm.

Comments

This string search routine returns results consistent with the Search function also in TPSTRING. However, BMSearch uses the Boyer-Moore algorithm, which in some cases leads to significantly improved performance compared to Search.

BMSearch differs from Search in the parameters that it requires. Like Search, it will search any Buffer region, with a BufLength of up to 65534 bytes. Unlike Search, it requires a special table (BT) previously initialized by BMMakeTable, and the search string is a normal Turbo string.

BMSearch returns the number of bytes searched to find the target string. If the string is not found, BMSearch returns $FFFF. If the target string is found at the very start of Buffer, BMSearch returns 0.

The performance of Boyer-Moore searching is dependent on many factors. If the buffer region is the size of a normal Turbo string (255 bytes or less), and the link table BT must be regenerated for each search, BMSearch will always be slower than the simpler Search routine. On the other hand, if the buffer is large, or if the same link table can be used many times, BMSearch's performance ranges from about the same as Search's to twice as fast, depending on the length of the MatchString. Typically, the longer the MatchString, the more advantage BMSearch will gain.

Special thanks to Don Strenczewilk for providing the initial version of the Boyer-Moore search routines.

Example

```
function BMS(var Buffer; BufLength : Word; MatchSt : string) : Word;
var
  LocalTable : BTable;
begin
  BMMakeTable(MatchSt, LocalTable);
  BMS := BMSearch(Buffer, BufLength, LocalTable, MatchSt);
end;
```

This function hides the link table from the calling program. Unfortunately, it also updates the link table at every call, thus minimizing the performance of the B-M search. This function will provide good performance only if the Buffer region is large. Otherwise, an application should call BMMakeTable explicitly and use the returned link table as many times as possible before discarding it.

See Also

BMMakeTable
BMSearchUC
Search

BMSearchUC

Declaration

```
function BMSearchUC(var Buffer; BufLength : Word;
                    BT : BTable; MatchString : string) : Word;
```

Purpose

Case-insensitive search of Buffer for MatchString using the Boyer-Moore algorithm.

Comments

This function operates just like BMSearch, except that searching is not case sensitive. See BMSearch for more details.

Important requirement: It is the application's responsibility to assure that MatchString is raised to uppercase prior to calling either BMMakeTable or BMSearchUC.

BMSearchUC almost always outperforms SearchUC, even if the link table BT must be built for each search. The speed of BMSearchUC ranges from slightly slower than SearchUC to about four times faster depending on how often the link table is rebuilt, the length of the MatchString, and the data contents.

Note that BMSearchUC handles correctly all alphabetics in the international character set ('A'..'Z', 'a'..'z', #128..#154, #160..#167).

Example

```
function UCBMS(var Buffer; BufLength : Word; MatchSt : string) : Word;
var
  LocalTable : BTable;
begin
  MatchSt := StUpCase(MatchSt);      {Guarantee that MatchSt is uppercase}
  BMMakeTable(MatchSt, LocalTable); {Build the link table}
  UCBMS := BMSearchUC(Buffer, BufLength, LocalTable, MatchSt);
end;
```

This function hides from the calling program all operations preparatory to a case-insensitive Boyer-Moore search. If the link table could be used more than once, this function will waste performance, although it will still be faster than most alternative approaches.

See Also

BMMakeTable
BMSearch
SearchUC

Center / CenterCh

Declaration

```
function Center(S : string; Width : Byte) : string;
```

Purpose

Return a string centered in a blank string of specified width.

Comments

If the length of S is greater than Width, then S is returned unchanged. If S is an empty string, a string of blanks is returned.

Example

```
Center('hello', 10)
```

Returns the string ' hello '

See Also

CenterCh

Declaration

```
function CenterCh(S : string; Ch : Char; Width : Byte) : string;
```

Purpose

Return a string centered in a string of Ch with specified width.

Comments

If the length of S is greater than Width, then S is returned unchanged. If S is an empty string, a string of characters Ch is returned.

Example

```
CenterCh('hello', '-', 10)
```

Returns the string '--hello---'

See Also

Center

CharStr / CleanPathName

Declaration

```
function CharStr(Ch : Char; Len : Byte) : string;
```

Purpose

Return a string of length Len filled with Ch.

Comments

If Len is zero, an empty string is returned.

Example

```
CharStr('-',10)
```

Returns '----------'

See Also

CenterCh

PadCh

Declaration

```
function CleanPathName(PathName : string) : string;
```

Purpose

Return a pathname cleaned up as DOS will do it.

Comments

You may have noticed that DOS will accept filenames containing more than eight characters in the name, or more than three characters in the extension. It simply truncates any excess characters. The same behavior applies to subdirectory names, whether given alone or as part of a pathname.

If your application displays filenames, it will look silly to display characters ignored by DOS. CleanPathName automatically truncates any extra characters.

Examples

```
CleanPathName('AUTOEXEC.BAT')
```

Returns 'AUTOEXEC.BAT'

```
CleanPathName('AUTOEXECA.BAT1')
```

Returns 'AUTOEXEC.BAT'

```
CleanPathName('D:\TEST12345\BUG.PAS')
```

Returns 'D:\TEST1234\BUG.PAS'

See Also

FullPathName

CompareLetterSets

Declaration

```
function CompareLetterSets(Set1, Set2 : LongInt) : Word;
```

Purpose

Returns the sum of the values of the letters common to Set1 and Set2.

Comments

When used in conjunction with MakeLetterSet, CompareLetterSets provides an alternative to the Soundex algorithm for judging the similarity of two words or strings of words. The algorithm was developed by Michael Bickel, and it is discussed at length in Jim Howell's "An Alternative to Soundex" (*Dr. Dobb's Journal*, November 1987).

Before calling CompareLetterSets, you must use MakeLetterSet to create a bit-mapped long integer (or "letter set") indicating what letters are used in a particular string; this process should be repeated for each string to be compared. Given two such sets, CompareLetterSets will then find the union of the two sets (the letters shared in common) and return a value that indicates how closely the two strings match. In calculating this value, it relies on the LetterValues table (declared in TPSTRING.PAS), which assigns relatively high values to rarely used letters ('J', 'Q', 'X', 'Z') and low ones to the most frequently used letters ('A', 'E', 'N', 'S', 'T'). The return value is simply the sum of the values for each set in the union. To determine the maximum possible value that could be returned in any given situation, you would compare the letter set for a string to itself:

```
PossibleValue := CompareLetterSets(LetterSet1, LetterSet1);
```

Comparisons to the letter sets for other strings will yield a result in the range 0..PossibleValue. The higher the value, the higher the degree of similarity.

Unfortunately, there is no definitive method for deciding whether or not the value returned by CompareLetterSets indicates a close similarity, so you'll probably need to run some experiments to determine how best to use these two functions. One way to make its results more informative would be to use a function such as DissimilarityIndex, shown in the example below, which returns a value between 0.0 (high degree of similarity) and 130.0 (first string is empty). (Similar strings will usually have an index of 0.3 or less.) You might also wish to modify the function to take into account factors that Bickel's algorithm ignores (are the lengths of the two strings the same? are the first characters the same? etc.). For a more complete discussion of the algorithm, see Howell's article in *Dr. Dobb's*.

CompareLetterSets

Example

```
function DissimilarityIndex(S1, S2 : string) : real;
var Set1, Set2 : LongInt;
    Possible, Actual, Delta : Integer;
begin
  Set1 := MakeLetterSet(S1);
  Set2 := MakeLetterSet(S2);
  Possible := CompareLetterSets(Set1, Set1);
  Actual := CompareLetterSets(Set1, Set2);
  Delta := Possible - Actual;
  if Possible = 0 then
    DissimilarityIndex := 130
  else
    DissimilarityIndex := Abs(Delta / Possible);
end;

var S1, S2 : string;
begin
  repeat
    {get first string}
    Write('String 1: '); ReadLn(S1);
    {get second string}
    Write('String 2: '); ReadLn(S2); Set2 := MakeLetterSet(S2);
    {show result}
    WriteLn('Index:  ', DissimilarityIndex(S1, S2):1:3);
  until False; {press ^C to stop}
end.
```

See Also

MakeLetterSet

Soundex

Declaration

```
function CompString(S1, S2 : string) : CompareType;
```

Purpose

Return less, equal, greater if s1<s2, s1=s2, or s1>s2.

Comments

CompString does the same thing as Turbo Pascal's string comparison infix operators, with one advantage. If an application branches to any of three statements depending on whether the comparison shows "equal", "greater", or "less", CompString will make the determination in a single comparison, while the infix operators will require at least two comparisons. Thus, for certain programs, CompString will offer a speed advantage.

Examples

```
var
  l1, l2 : string;
case CompString(l1, l2) of
  Less :   writeln('l1 < l2');
  Equal:   writeln('l1 = l2');
  Greater: writeln('l1 > l2');
end;
```

This general structure may be used when three different outcomes hinge on a single string comparison.

```
CompString('aaa', 'aaaa')
```

Returns Less.

```
CompString('aaab', 'aaaa')
```

Returns Greater.

```
CompString('Aaaa', 'aaaa')
```

Returns Less.

See Also

CompUCString

CompStruct

Declaration

```
function CompStruct(var S1, S2; Size : Word) : CompareType;
```

Purpose

Compare two fixed-size structures.

Comments

Pascal does not provide built-in support for comparison of structured data types, such as arrays or records. In some cases this is a sensible restriction. For example, consider a record containing an integer followed by a string. A comparison routine would have no reasonable way of knowing that a string was embedded, and as a result any garbage characters beyond the current length of the string would play a role in the comparison.

In spite of these fundamental difficulties, comparison of structures is often desirable. The CompStruct function offers a solution. Although CompStruct returns a result from the enumerated type Less, Equal, Greater, in most cases only equality or non-equality will be significant. CompStruct uses ASCII comparison on a byte-by-byte basis to determine Less or Greater status.

Note that the problem regarding embedded strings applies to CompStruct. It is the programmer's responsibility to avoid this situation.

The Size parameter to CompStruct is the number of bytes to compare. The best way to determine Size is with the built-in Turbo operator SizeOf().

Examples

```
type
  ArrayType = array[1..3] of byte;
const
  A1 : ArrayType = (0, 1, 2);
  A2 : ArrayType = (0, 1, 3);
  A3 : ArrayType = (0, 1, 2);

CompStruct(A1, A2, SizeOf(ArrayType))
```

Returns Less.

```
CompStruct(A1, A3, SizeOf(ArrayType))
```

Returns Equal.

See Also

CompString

Declaration

```
function CompUCString(S1, S2 : string) : CompareType;
```

Purpose

Compare two strings in a case-insensitive manner.

Comments

CompUCString converts lowercase characters to uppercase prior to comparing two strings. Although slower than CompString or Turbo's built-in string comparison, CompUCString is much faster than calling StUpCase and then performing a normal string comparison. Note that CompUCString handles correctly all alphabetics in the international character set ('A'..'Z', 'a'..'z', #128..#154, #160..#167).

Examples

```
CompUCString('AAA', 'aaaa')
```

Returns Less.

```
CompUCString('AAAB', 'aaaa')
```

Returns Greater.

```
CompUCString('Aaaa', 'aaaa')
```

Returns Equal.

See Also

CompString

Declaration

```
function DefaultExtension(Name, Ext : string) : string;
```

Purpose

Return a filename or pathname with a default extension attached.

Comments

If Name already has an extension, no change is made. Otherwise, Ext is appended to it. Ext should be a maximum of three characters and *not* begin with a period.

Examples

```
DefaultExtension('TEST', 'pas')
```

Returns 'TEST.pas'

```
DefaultExtension('TEST.', 'PAS')
```

Returns 'TEST.'

See Also

ForceExtension

DeTab

Declaration

```
function DeTab(S : string; TabSize : Byte) : string;
```

Purpose

Expand tabs in string S to blanks on spacing TabSize.

Comments

DeTab removes tabs from a string, converting them into appropriate strings of blanks.

Unlike EnTab, DeTab will apply its conversion to all parts of a string, whether or not the string contains quoted literals.

A TabSize of 8 is the normal size used by most text editors. If TabSize is zero, DeTab will always return an empty string.

Example

```
var
  S : string;
begin
  while not EOF(Input) do begin
    ReadLn(Input, S);
    WriteLn(Output, DeTab(S, 8));
  end;
end.
```

Implements a complete filter for expanding tabs in a text file.

See Also

EnTab

Declaration

```
procedure DisposeString(P : Pointer);
```

Purpose

Deallocate space for string at p.

Comments

Applications that perform extensive text processing are often better off storing strings on the heap rather than as global variables. The heap offers two advantages in this situation: it can grow to use all 640K of DOS memory, and strings can use just the memory they need rather than the maximum possible length. For example, suppose a filter program must read in each line of a text file and store it in memory prior to processing. By storing each line as global data, even one character would consume the maximum line length. By storing the lines on the heap, each string would consume exactly what it needed.

DisposeString works in concert with StringToHeap. StringToHeap allocates heap space for a specified string and returns a pointer to the string. DisposeString returns the heap space associated with that pointer to the free memory pool.

If the pointer passed to DisposeString is nil, no action is performed.

Example

```
var
  P : pointer;
  S : string;
begin
  P := StringToHeap('hello');
  S := StringFromHeap(P);
  DisposeString(P);
end;
```

The first StringToHeap statement allocates 6 bytes of heap space for 'hello' plus its length byte. StringFromHeap loads 'hello' into S. DisposeString releases the initial 6 bytes of heap space.

See Also

StringFromHeap

StringToHeap

EnTab

Declaration

```
function EnTab(S : string; TabSize : Byte) : string;
```

Purpose

Convert blanks in string S to tabs on spacing TabSize.

Comments

EnTab compresses a string by converting runs of spaces to appropriate sequences of tabs and spaces.

EnTab recognizes literal strings delimited by single quotes or double quotes, and does not convert spaces to tabs within these strings. It also recognizes the C "escape" character '\' to allow quotes embedded within strings.

A TabSize of 8 is the normal size used by most text editors. If TabSize is zero, Entab will always return an empty string.

Example

```
var
  S : string;
begin
  while not EOF(Input) do begin
    ReadLn(Input, S);
    WriteLn(Output, EnTab(S, 8));
  end;
end.
```

Implements a complete filter for compressing text files by tab conversion.

See Also

DeTab

Declaration

```
function ExtractWord(N : Byte; S : string; WordDelims : CharSet) : string;
```

Purpose

Given a set of word delimiters, return the N'th word in S.

Comments

This routine returns the N'th word in the specified string. WordDelims is a user-defined set of word delimiters used to determine where words begin and end. At the very least, this set should contain ' '. If S contains fewer than N words, the string returned will be empty.

Example

```
S := 'Four score and seven years ago, our forefathers...';
SixthWord := ExtractWord(6, S, [' ', ',', '.']);
```

The result would be 'ago', since ',' is a word delimiter.

See Also

WordCount

Declaration

```
function ForceExtension(Name, Ext : string) : string;
```

Purpose

Force the specified extension onto the file name.

Comments

Whether or not the given name has an extension, ForceExtension returns a new filename with the specified extension.

Examples

```
ForceExtension('TEST', 'pas')
```

Returns 'TEST.pas'

```
ForceExtension('TEST.', 'PAS')
```

Returns 'TEST.PAS'

```
ForceExtension('C:\AUTOEXEC.BAT', 'PAS')
```

Returns 'C:\AUTOEXEC.PAS'

See Also

DefaultExtension

Form

Declaration

```
function Form(Mask : string; R : Float) : string;
```

Purpose

Returns a formatted string with digits from R merged into the Mask.

Comments

Form provides the equivalent of Turbo-BCD's Form function for 6-byte or 8087 reals in Turbo Pascal version 4.0 or higher. The Form function offers advanced numeric formatting especially applicable to financial reports.

Since Pascal does not allow user-defined routines to accept a variable number of parameters, the Turbo Professional Form function accepts only a single parameter of type Real.

The Mask parameter is a string that may contain any combination of normal characters as well as a sequence of the special characters # * @ $ - + , . This sequence is called a numeric field, and Form will replace it with a formatted version of the real number. The number is returned right-justified within the field, decimals are returned rounded if they exceed the number of decimals specified by the format, and if the number is too large to be returned in the field, all digit positions are filled with asterisks.

The special characters are interpreted as follows:

A digit position. If the numeric field contains no * or @ characters, unused digits are returned as blanks. If the numeric field contains no sign positions (+ or - characters) and the number is negative, a floating minus is returned in front of the number. Examples:

```
Form('####', 34.567) returns '  35'
Form('####.##', -12.3) returns ' -12.30'
Form('##.##', 1000.0) returns '**.**'
```

* A digit position. Unused positions are returned as asterisks instead of blanks. The * character needs only occur once in the numeric field to activate this effect. The sign of the number will not be returned unless the field also contains a sign character (- or +). Examples:

```
Form('*##.#', 4.567) returns '**4.57'
Form('****', 123) returns '*123'
```

@ A digit position. Unused positions are returned as zeros instead of blanks. The @ character needs only occur once in the numeric field to activate this effect. The sign of the number will not be returned unless the field also contains a sign character (- or +). Examples:

```
Form('@##', 9) returns '009'
Form('@@@.@@', 12.3) returns '012.30'
```

$ A digit position. A floating dollar sign is returned in front of the number. The $ character needs only occur once in the numeric field to activate this effect. Examples:

```
Form('$####.##', 123.45) returns ' $123.45'
Form('*$####.##', 12.34) returns '***$12.34'
```

- A sign position. If the number is negative, a minus will be returned in that position. If it is positive, a blank will be returned. Examples:

```
Form('-###.##', -1.2) returns '-  1.20'
Form('-###.##', 1.2) returns '   1.20'
```

+ A sign position. If the number is positive, a plus will be returned in that position. If it is negative, a minus will be returned. Examples:

```
Form('+###.##', 1.2) returns '+  1.20'
Form('*$####.##+', -12.34) returns '***$12.34-'
```

, A decimal comma or a separator comma.

. A decimal period or a separator period. The last period or comma in the numeric field is considered the decimal delimiter. Examples:

```
Form('##,###,###.##', 12345678.91) returns '12,345,678.91'
Form($#.###.###,##', -12345.6) returns '  -$12.345,60'
```

Note: Several users of earlier versions of Turbo Professional reported what they thought was a bug in Form. Consider the following:

```
WriteLn( Form('###,###,###', 1000.0) );
```

What they expected was ' 1,000'. What they got was ' 1,000,000'. This is *not* a bug. It is consistent with the behavior of the Form function in TURBOBCD 3.0, which treated the last period *or comma* in the mask as marking the decimal point, allowing Form to be used in countries that use commas in places where Americans use periods.

See Also

Form (TPBCD)

FullPathName

Declaration

```
function FullPathName(FName : string) : string;
```

Purpose

Given FName (known to exist), return a full pathname.

Comments

FullPathName does not search the DOS PATH to find a file (see ExistOnPath for that operation). Instead, FullPathName returns a complete DOS pathname from a partial one that already allows DOS to find the file. The pathname will include a drive letter, complete directory, and filename.

In many cases, FullPathName must interrogate DOS to determine the current directory on the current or specified drive. For this purpose, a disk must have been loaded into the drive or a DOS critical error will occur.

Examples

```
FullPathName('AUTOEXEC.BAT')
```

Returns 'C:\AUTOEXEC.BAT' if the current drive is C:

```
FullPathName('D:..\TEST\CHECK.PAS')
```

Returns 'D:\EXAMPLES\..\TEST\CHECK.PAS' if the current directory on drive D: is \EXAMPLES

See Also

ExistOnPath (TPDOS)

Declaration

```
function HexB(B : Byte) : string;
```

Purpose

Return hex string for Byte.

Comments

Returns a 2-character hexadecimal string for the byte. Leading zeros are included. No '$' or 'h' characters are included.

Examples

```
HexB(0)
```

Returns '00'

```
HexB(250)
```

Returns 'FA'

See Also

HexL HexW

Declaration

```
function HexL(L : LongInt) : string;
```

Purpose

Return hex string for LongInt.

Comments

Returns an 8-character hexadecimal string for the LongInt. Leading zeros are included. No '$' or 'h' characters are included.

Examples

```
HexL(0)
```

Returns '00000000'

```
HexL(655360)
```

Returns '000A0000'

See Also

HexB HexW

HexPtr / HexW

Declaration

```
function HexPtr(P : Pointer) : string;
```

Purpose

Return hex string for pointer.

Comments

Given a pointer, HexPtr returns a string of the form 'ssss:oooo', where 'ssss' is the segment portion of the address, and 'oooo' is the offset.

Example

```
if (ErrorAddr <> nil) or (ExitCode <> 0) then begin
  {don't report ^Break as an error}
  if ExitCode <> $FF then
    WriteLn('Error ', HexW(ExitCode), ' at ', HexPtr(ErrorAddr));
  {prevent runtime library from reporting error again}
  ErrorAddr := nil;
  ExitCode := 0;
end;
```

Code such as the above might be included in a user-defined exit handler to report runtime errors. HexPtr is used to turn the address of the error into a string.

Declaration

```
function HexW(W : Word) : string;
```

Purpose

Return hex string for Word.

Comments

Returns a 4-character hexadecimal string for the word. Leading zeros are included. No '$' or 'h' characters are included.

Examples

```
HexW(0)
```

Returns '0000'

```
HexW(MaxInt)
```

Returns '7FFF'

See Also

HexB

HexL

JustExtension / JustFilename

Declaration

```
function JustExtension(Name : string) : string;
```

Purpose

Return just the extension of a pathname.

Comments

Drive, directory and filename are stripped from a path, returning just the extension. The extension does not include a period. If the input string has no extension, the empty string is returned.

Examples

```
JustExtension('TEST.PAS')
```

Returns 'PAS'

```
JustExtension('C:\DATAF')
```

Returns empty string ''

See Also

JustFilename JustPathname

Declaration

```
function JustFilename(PathName : string) : string;
```

Purpose

Return just the filename and extension of a pathname.

Comments

Drive letter and directory are stripped from the input string. Pathname is generally expected to be a complete pathname, with both a path and a filename. If there is no ':' or '\' in PathName, the input string will be returned unchanged.

Examples

```
JustFilename('C:\AUTOEXEC.BAT')
```

Returns 'AUTOEXEC.BAT'

```
JustFilename('test.')
```

Returns 'test.'

See Also

JustExtension JustPathname

JustPathname

Declaration

```
function JustPathname(PathName : string) : string;
```

Purpose

Return just the drive:directory portion of a pathname.

Comments

The filename and extension of the input string are stripped. Pathname is generally expected to be a complete pathname, with both a path and a filename. If there is no ':' or '\' in PathName, the input string will be returned unchanged.

Examples

```
JustPathname('C:\AUTOEXEC.BAT')
```

Returns 'C:\'

```
JustPathname('..\test\bug.pas')
```

Returns '..\test'

See Also

JustExtension

JustFilename

Declaration

```
function LeftPad(S : string; Len : Byte) : string;
```

Purpose

Return a string left-padded to length Len with blanks.

Comments

Blanks are inserted to the left of the input string until the total length equals Len. If S is longer than Len, then S is returned unchanged. If S is the empty string, a string of blanks is returned.

Examples

```
LeftPad('hello', 7)
```

Returns ' hello'

```
LeftPad('gargantuan', 5)
```

Returns 'gargantuan'

See Also

Center Pad

Declaration

```
function LeftPadCh(S : string; Ch : Char; Len : Byte) : string;
```

Purpose

Return a string left-padded to length Len with Ch.

Comments

Characters are inserted to the left of the input string until the total length equals Len. If S is longer than Len, then S is returned unchanged. If S is the empty string, a string of Ch is returned.

Examples

```
LeftPadCh('hello', '-', 7)
```

Returns '--hello'

```
LeftPadCh('gargantuan', 5)
```

Returns 'gargantuan'

See Also

Center Pad

LoCase / LoCaseMac

Declaration

```
function LoCase(Ch : Char) : Char;
```

Purpose

Return Ch in lower case.

Comments

This routine converts characters to lower case. Unlike LoCaseMac, LoCase correctly handles all alphabetics in the international character set ('A'..'Z', 'a'..'z', #128..#154, #160..#167).

Examples

```
LoCase('A')
```

Returns 'a'

```
LoCase(#165) {'N' with a tilde ('~')}
```

Returns #164 ('n' with a tilde).

See Also

StLoCase

UpCase

Declaration

```
function LoCaseMac(Ch : Char) : Char;
```

Purpose

Return Ch in lower case.

Comments

An inline macro that converts characters to lower case as rapidly as possible.

Examples

```
LoCaseMac('A')
```

Returns 'a'

```
LoCaseMac('%')
```

Returns '%'

See Also

UpCaseMac

Declaration

```
function Long2Str(L : LongInt) : string;
```

Purpose

Convert a LongInt/Word/Integer/Byte/ShortInt to a string.

Comments

This routine calls Turbo's built-in Str procedure and then assigns the result to the function. The resulting string has no leading or trailing blanks.

Turbo's automatic type promotion allows bytes, integers or words to be used as the parameter to this routine.

Examples

```
Long2Str(1000)
```

Returns '1000'

```
Long2Str(-1)
```

Returns '-1'

See Also

Str2Int

Declaration

```
function MakeLetterSet(S : string) : LongInt;
```

Purpose

Returns a bit-mapped LongInt storing the individual letters contained in S.

Comments

MakeLetterSet creates a bit-mapped set (in the form of a LongInt) indicating which letters of the alphabet are in S. For example, if the string were 'A' (or 'a'), MakeLetterSet would return $00000001; if it were 'ABC', the result would be $00000007, and so on. The maximum possible value of the result is $003FFFFF (first 26 bits set). See the discussion of CompareLetterSets for more details.

See Also

CompareLetterSets

OctalB / OctalL

Declaration

```
function OctalB(B : Byte) : string;
```

Purpose

Return octal string for Byte.

Comments

Returns a 3-character string in octal (base 8) format. Leading zeros are included.

Examples

```
OctalB(0)
```

Returns '000'

```
OctalB(255)
```

Returns '377'

See Also

OctalW

Declaration

```
function OctalL(L : LongInt) : string;
```

Purpose

Return octal string for LongInt.

Comments

Returns a 12-character string in octal (base 8) format. Leading zeros are included.

Examples

```
OctalL(0)
```

Returns '000000000000'

```
OctalL(655360)
```

Returns '000002400000'

See Also

OctalW

Declaration
```
function OctalW(W : Word) : string;
```

Purpose
Return octal string for Word.

Comments
Returns a 6-character string in octal (base 8) format. Leading zeros are included.

Examples
```
OctalW(0)
```
Returns '000000'
```
OctalW(65535)
```
Returns '177777'

See Also
OctalB

OctalL

Declaration
```
function Pad(S : string; Len : Byte) : string;
```

Purpose
Return a string right-padded to length Len with blanks.

Comments
Blanks are added to the end of the input string until its length equals Len. If the length of S already equals or exceeds Len, S is returned unchanged. If S is the empty string, a string of blanks is returned.

Examples
```
Pad('hello', 8)
```
Returns 'hello '
```
Pad('gargantuan', 8)
```
Returns 'gargantuan'

See Also
Center

LeftPad

PadCh

PadCh / Real2Str

Declaration

```
function PadCh(S : string; Ch : Char; Len : Byte) : string;
```

Purpose

Return a string right-padded to length Len with Ch.

Comments

Characters are added to the end of the input string until its length equals Len. If the length of S already equals or exceeds Len, S is returned unchanged. If S is the empty string, a string of Ch is returned.

Examples

```
PadCh('hello', '-', 8)
```

Returns 'hello---'

```
PadCh('gargantuan', '-', 8)
```

Returns 'gargantuan'

See Also

Pad

Declaration

```
function Real2Str(R : Float; Width, Places : Byte) : string;
```

Purpose

Convert a real to a string.

Comments

This routine calls Turbo's built-in Str procedure to convert R to a string of length Width, which is returned as the function result. Places is the number of decimal places to round to.

Note: If TPSTRING is compiled with $N+, this routine works with variables of type Extended, else with variables of type Real.

Example

```
R := 1234.56789;
Str(R:7:2, S);
S := Real2Str(R, 7, 2);
```

In both cases, S would be set to '1234.57'.

Declaration

```
function Search(var Buffer; BufLength : Word;
                var Match; MatLength : Word) : Word;
```

Purpose

Search through Buffer for Match.

Comments

This is a very fast general purpose search function that applies equally well to strings or longer byte sequences.

The variable to be searched is passed in the Buffer parameter. The length of the region to be searched is passed in BufLength. The string to match is passed in Match. The number of characters to match is passed in MatLength.

If the Match string is found, Search returns the number of characters that were searched to reach the beginning of the Match string. Note that this may be zero or more characters. If the Match string is not found, the value $FFFF is returned.

Since the two string buffers are VAR parameters, literal strings may not be used as parameters.

Examples

```
var
  S, T : string;
  Posn : Word;
begin
  S := 'Now is the time for the show to start';
  T := 'Now';
  Posn := Search(S[1], Length(S), T[1], Length(T)); {Posn is 0}
  T := 'the';
  Posn := Search(S[0], Length(S), T[1], Length(T)); {Posn is 8}
  Posn := Search(S[Posn+1], Length(S)-Posn, T[1], Length(T)); {Posn is 13}
  T := 'zzz';
  Posn := Search(S[0], Length(S), T[1], Length(T)); {Posn is $FFFF}
end.
```

Starting the search at position 0 of the string makes Search return match values just like those of Turbo's Pos function. Starting at position zero may be dangerous though, since the length byte will be included in the characters searched. By indexing the Buffer string as shown in the example, previous matches may be skipped.

See Also

BMSearch

PosAsc (TPASCIIZ)

SearchUC

Declaration

```
function SearchUC(var Buffer; BufLength : Word;
                  var Match; MatLength : Word) : Word;
```

Purpose

Case-insensitive search through Buffer for a Match.

Comments

The function works in a manner identical to Search (TPSTRING) except that the comparison is made in a case-insensitive manner. Characters in both the Buffer and the Match strings are raised to uppercase prior to comparison. The uppercasing does not change the actual parameters.

If the Match string is not found, SearchUC returns $FFFF. Otherwise it returns the number of characters searched to find the start of the match string.

SearchUC runs 2-4 times slower than Search, due to the uppercasing operation.

See the Search function in TPSTRING for further information and examples.

Note that SearchUC handles correctly all alphabetics in the international character set ('A'..'Z', 'a'..'z', #128..#154, #160..#167).

See Also

Search

BMSearchUC

Declaration

```
function Soundex(S : string) : string;
```

Purpose

Return 4-character soundex of input string.

Comments

The soundex code is often used to provide approximate matching of proper names. Soundex returns a 4-character string encoding the input string with the soundex algorithm. The first character is just the uppercase value of the input string's first character. The remaining three characters are numeric digits that encode the *sound* of the input string. Comparing the soundex strings of two similar names will usually yield an exact match.

Soundex is useful only for strings comprised mostly of alphabetic characters. Leading blanks should be removed prior to calling Soundex. Soundex internally raises the string to uppercase.

Examples

```
Soundex('macintosh')
```

Returns 'M253'

```
Soundex('Mcntoch')
```

Returns 'M253'

Declaration

```
function StLoCase(S : string) : string;
```

Purpose

Convert upper case letters in string to lowercase. Note that StLoCase handles correctly all alphabetics in the international character set ('A'..'Z', 'a'..'z', #128..#154, #160..#167).

Examples

```
StLoCase('ABC')
```

Returns 'abc'

```
StLoCase('!@# rats!')
```

Returns '!@# rats!'

See Also

LoCase

StUpCase

Str2Int / Str2Long

Declaration

```
function Str2Int(S : string; var I : integer) : Boolean;
```

Purpose

Convert a string to an integer, returning true if successful.

Comments

This routine calls the built-in Turbo Val procedure, assigning the numeric result to the VAR parameter, and the success code to a boolean function result. Any trailing blanks are stripped from S before Val is called.

If Str2Int returns False, the VAR parameter I will contain the position within S that caused the error. If the input string begins with '$', the string is interpreted as a hexadecimal number. Leading blanks are acceptable.

Examples

```
Str2Int(' 123', I)
```

Returns True and I = 123

```
Str2Int('$00F0', I)
```

Returns True and I = 240

```
Str2Int('12z', I)
```

Returns False and I = 3

Declaration

```
function Str2Long(S : string; var I : LongInt) : Boolean;
```

Purpose

Convert a string to a LongInt, returning True if successful.

Comments

This routine calls the built-in Turbo Val procedure, assigning the numeric result to the VAR parameter, and the success code to a boolean function result. Any trailing blanks are stripped from S before Val is called.

If Str2Long returns False, the VAR parameter I will contain the position within S that caused the error. If the input string begins with '$', the string is interpreted as a hexadecimal number. Leading blanks are acceptable.

Example

```
Str2Long('$A0000', I)
```

Returns True and I = 655360

See Also

Long2Str

Declaration

```
function Str2Real(S : string; var R : Float) : Boolean;
```

Purpose

Convert a string to a real, returning True if successful.

Comments

This routine calls the built-in Turbo Val procedure, assigning the numeric result to the VAR parameter and the success code to a boolean function result. Trailing blanks are trimmed before calling Val. Under Turbo 4.0, Str2Real also insures that Val will accept numbers like '5.', '.3', and '-.1'. If Str2Real returns False, the VAR parameter R will contain the position within S that caused the error.

Note: If TPSTRING is compiled with $N+, this routine works with variables of type Extended, else with variables of type Real.

Example

```
Str2Real(' -.123 ', R);
```

Returns True and R = -0.123.

```
Str2Real('123. 4567', R);
```

Returns False and R = 5.0.

Declaration

```
function Str2Word(S : string; var I : Word) : Boolean;
```

Purpose

Convert a string to a word, returning True if successful.

Comments

This routine calls the built-in Turbo Val procedure, assigning the numeric result to the VAR parameter, and the success code to a boolean function result. Any trailing blanks are stripped from S before Val is called.

If Str2Word returns False, the VAR parameter I will contain the position within S that caused the error. If the input string begins with '$', the string is interpreted as a hexadecimal number. Leading blanks are acceptable.

Examples

```
Str2Word(' 123', I)
```

Returns True and I = 123

```
Str2Word('12z', I)
```

Returns False and I = 3

See Also

Long2Str

StringFromHeap

Declaration

```
function StringFromHeap(P : Pointer) : string;
```

Purpose

Return string at P^.

Comments

Applications that perform extensive text processing are often better off storing strings on the heap rather than as global variables. The heap offers two advantages in this situation: it can grow to use all 640K of DOS memory, and strings can take just the memory they need rather than using the maximum possible length. For example, suppose a filter program must read in each line of a text file and store it in memory prior to processing. By storing each line as global data, even one character consumes the maximum line length. By storing the lines on the heap, each string consumes exactly what it needs.

StringFromHeap works in concert with StringToHeap. StringToHeap allocates heap space for a specified string and returns a pointer to the string. StringFromHeap returns the actual string from the heap for further use in an expression.

If the pointer passed to StringFromHeap is nil, the empty string is returned. StringFromHeap may be used in string expressions.

Example

```
var
  P : pointer;
  S : string;
begin
  P := StringToHeap('PASFILE.INC');
  S := 'C:\'+StringFromHeap(P);
end;
```

StringToHeap allocates 12 bytes of heap space for 'PASFILE.INC' plus its length byte. StringFromHeap loads 'C:\PASFILE.INC' into S.

See Also

DisposeString

StringToHeap

Declaration

```
function StringToHeap(S : string) : Pointer;
```

Purpose

Allocate space for S and return pointer.

Comments

Applications that perform extensive text processing are often better off storing strings on the heap rather than as global variables. The heap offers two advantages in this situation: it can grow to use all 640K of DOS memory, and strings can take just the memory they need rather than using the maximum possible length. For example, suppose a filter program must read in each line of a text file and store it in memory prior to processing. By storing each line as global data, even one character consumes the maximum line length. By storing the lines on the heap, each string consumes exactly what it needs.

StringToHeap works in concert with StringFromHeap and DisposeString. StringToHeap allocates heap space for a specified string and returns a pointer to the string. StringFromHeap returns the actual string from the heap for further use in an expression.

If insufficient heap space remains to store a string, StringToHeap returns a nil pointer.

Example

See the entry for StringFromHeap.

See Also

DisposeString

StringFromHeap

StUpCase / Trim

Declaration

```
function StUpCase(S : string) : string;
```

Purpose

Convert lower case letters in string to uppercase.

Comments

Note that StUpCase handles correctly all alphabetics in the international character set ('A'..'Z', 'a'..'z', #128..#154, #160..#167).

Examples

```
StUpCase('hello')
```

Returns 'HELLO'

```
StUpCase('!@#$ drat!')
```

Returns '!@#$ DRAT!'

See Also

StLoCase UpCase

Declaration

```
function Trim(S : string) : string;
```

Purpose

Return a string with leading and trailing white space removed.

Comments

White space is defined as any ASCII character whose ordinal value is 32 or less. White space internal to the string S is returned unchanged. If the input string is composed entirely of white space, the empty string is returned.

Examples

```
Trim('  hello  ')
```

Returns 'hello'

```
Trim(^I^I'hello'^M^J)
```

Returns 'hello'

See Also

TrimLead TrimTrail

Declaration

```
function TrimLead(S : string) : string;
```

Purpose

Return a string with leading white space removed.

Comments

White space is defined as any ASCII character whose ordinal value is 32 or less. White space internal to the string S is returned unchanged. If the input string is composed entirely of white space, the empty string is returned.

Examples

```
TrimLead('  hello  ')
```

Returns 'hello '

```
TrimLead(^I^I'hello'^M^J)
```

Returns 'hello'^M^J

See Also

Trim TrimTrail

Declaration

```
function TrimTrail(S : string) : string;
```

Purpose

Return a string with trailing white space removed.

Comments

White space is defined as any ASCII character whose ordinal value is 32 or less. White space internal to the string S is returned unchanged. If the input string is composed entirely of white space, the empty string is returned.

Examples

```
TrimTrail('  hello  ')
```

Returns ' hello'

```
TrimTrail(^I^I'hello'^M^J)
```

Returns ^I^I'hello'

See Also

Trim TrimLead

UpCase / UpCaseMac

Declaration

```
function UpCase(Ch : Char) : Char;
```

Purpose

Return Ch in upper case.

Comments

This routine converts characters to upper case. Unlike both the standard UpCase function in SYSTEM and UpCaseMac, UpCase correctly handles all alphabetics in the international character set ('A'..'Z', 'a'..'z', #128..#154, #160..#167).

Examples

```
UpCase('a')
```

Returns 'A'

```
UpCase(#164) {'n' with a tilde ('~')}
```

Returns #165 ('N' with a tilde).

See Also

LoCase StUpCase

Declaration

```
function UpCaseMac(Ch : Char) : Char;
```

Purpose

Return Ch in upper case.

Comments

An inline macro that converts characters to upper case as rapidly as possible.

Examples

```
UpCaseMac('a')
```

Returns 'A'

```
UpCaseMac('%')
```

Returns '%'

See Also

LoCaseMac

Declaration

```
function WordCount(S : string; WordDelims : CharSet) : Byte;
```

Purpose

Given a set of word delimiters, return number of words in S.

Comments

This routine returns the number of words in the specified string. WordDelims is a user-defined set of word delimiters used to determine where words begin and end. At the very least, this set should contain ' ', but in some applications you may want to use a larger set like the one in the example below.

Example

```
var
  S : string;
const
  WordDelims : CharSet =
    [#0..'"', '.', ',', ':', ';', '?', '(', ')', '[', ']', '{', '}'];
  N : Word = 0;
begin
  while not EOF(Input) do begin
    ReadLn(Input, S);
    Inc(N, WordCount(S, WordDelims));
  end;
  WriteLn(N, ' words in input file');
end.
```

A simple word counting program that reads from the standard input device.

See Also

ExtractWord

WordWrap

Declaration

```
procedure WordWrap(InSt : string; var OutSt, Overlap : string;
                   Margin : Byte; PadToMargin : Boolean);
```

Purpose

Wrap InSt at Margin, storing the result in OutSt and the remainder in Overlap.

Comments

This routine takes as input a string of text (InSt) and returns an output string (OutSt) no greater than Margin characters in length. If the margin falls in the middle of a word (defined as a series of characters bounded by either spaces or one end of the string), the output string will contain all previous words in the string, while all text from the word straddling the margin to the end of the string will be returned in Overlap. OutSt must therefore be large enough to hold a string of Margin characters, and Overlap must be at least as large as InSt.

If the input string cannot be word wrapped--that is, if Length(InSt) is greater than Margin, and there are no spaces between InSt[1] and InSt[Margin]--the output string will simply contain the first Margin characters in InSt.

PadToMargin indicates whether or not the output string should be padded with blanks to the specified Margin.

Example

```
var
  InputString, OutputString : string;
begin
  while not EOF(Input) do begin
    ReadLn(Input, InputString);
    InputString := Detab(InputString, 8);
    repeat
      WordWrap(InputString, OutputString, InputString, 40, False);
      WriteLn(Output, OutputString);
    until Length(InputString) = 0;
  end;
end.
```

A simple filter program for ASCII text files that expands all tabs in input lines and wraps output lines at column 40.

See Also

Detab

C. TPASCIIZ: Long ASCIIZ Strings

The TPASCIIZ unit defines a new string type for Turbo Pascal. Unlike normal Turbo strings, which are limited to 255 bytes in length, the ASCIIZ string type can be of any length up to 65520 characters.

The ASCIIZ string is defined in the same way as the one used by MS-DOS for internal operations and by the C programming language. An ASCIIZ is simply an array of characters terminated by a null character (ASCII #0).

Two disadvantages arise when using ASCIIZ strings. First, the length of such a string cannot be immediately determined. Instead, a length routine must scan the entire string until it encounters a null. TPASCIIZ implements this scan with a Turbo inline macro, and uses the 8088 SCASB instruction to obtain best possible performance for this operation. As a result, performance of the length operation should not be a practical problem.

The second disadvantage may rule out the use of ASCIIZ strings for some applications. Because a null character terminates an ASCIIZ, null characters may not be part of the string itself. Thus, ASCIIZ strings cannot contain extended scan code sequences (which begin with a null) unless the nulls are first translated into another character.

Although ASCIIZ strings may be as long as 65,520 characters, the TPASCIIZ unit defines a default maximum length of 1024 characters. This limit will allow you to declare a number of ASCIIZ strings as global variables without gobbling up the entire 64K segment. If you need even longer strings, change the constant MaxAsciiz. In this case, ASCIIZ strings should probably be allocated on the heap.

Constants

```
MaxAsciiz = 1024;
```

The maximum length of an Asciiz string. It may be increased up to 65,520.

```
NotFound = $FFFF;
```

Value returned by PosAsc and PosStr if the target substring is not found.

Types

```
Asciiz = array[0..MaxAsciiz] of Char;
```

The internal representation of an ASCIIZ string. One extra character position is allocated to store the null termination byte.

```
AsciizPtr = ^Asciiz;
```

A pointer to an ASCIIZ string. Because these strings are so large, it is often desirable to allocate them on the heap.

Asc2Str / AscCenter

Declaration

```
function Asc2Str(var A : Asciiz) : string;
```

Purpose

Convert Asciiz to Turbo string.

Comments

If the Asciiz exceeds 255 characters, it is truncated prior to assignment to the Turbo string.

Example

```
var
  A : Asciiz;
  S : string;
begin
  Str2Asc('hello', A);
  S := Asc2Str(A); {S gets 'hello'}
end.
```

See Also

Str2Asc

Declaration

```
procedure AscCenter(var A : Asciiz; Width : word; var B : Asciiz);
```

Purpose

Return an Asciiz centered in an Asciiz of blanks with specified width.

Comments

If A already exceeds Width, the returned Asciiz B is the same as A. Otherwise, B returns an Asciiz of specified Width with A centered in a field of blanks.

Example

```
var
  A, B : Asciiz;
begin
  Str2Asc('hello', A);
  AscCenter(A, 10, B); {B gets '   hello  '}
end.
```

See Also

AscCenterCh

Center (TPSTRING)

Declaration

```
procedure AscCenterCh(var A : Asciiz; Ch : Char; Width : word; var B : Asciiz);
```

Purpose

Return an Asciiz centered in an Asciiz of Ch with specified width.

Comments

If A already exceeds Width, the returned Asciiz B is the same as A. Otherwise, B returns an Asciiz of specified Width with A centered in a field of Ch.

Example

```
var
  A, B : Asciiz;
begin
  Str2Asc('hello', A);
  AscCenterCh(A, '-', 10, B); {B gets '---hello--'}
end.
```

See Also

AscCenter

CenterCh (TPSTRING)

Declaration

```
procedure AscCharStr(Ch : Char; Len : word; var A : Asciiz);
```

Purpose

Return an Asciiz of length Len filled with Ch.

Comments

If Len is zero, a null Asciiz is returned. Otherwise an Asciiz of length Len is returned, filled with character Ch.

Example

```
AscCharStr('-', 10, A)
```

Leaves A as '----------'

See Also

CharStr (TPSTRING)

AscFromHeap

Declaration

```
procedure AscFromHeap(P : AsciizPtr; var A : Asciiz);
```

Purpose

Return an Asciiz from the heap, empty if pointer is nil.

Comments

Because Asciiz strings can be so long, it is often desirable to store them on the heap rather than in the data segment where they can quickly consume the available 64K bytes. AscFromHeap works in conjunction with AscToHeap and DisposeAsc.

AscToHeap allocates heap space for an Asciiz and returns a pointer to it. AscFromHeap returns the data from the heap to another variable. DisposeAsc deallocates the heap space when it is no longer needed.

If the pointer P is nil, AscFromHeap returns an empty string.

Note that the statement

```
A := P^;
```

yields the same result as

```
AscFromHeap(P, A);
```

However, the call to AscFromHeap may be faster since it moves only the actual data rather than the total number of bytes in the structure.

Example

```
var
  P : AsciizPtr;
  A : Asciiz;
begin
  Str2Asc('hello', A);
  P := AscToHeap(A);
  AscFromHeap(P, A); {A gets 'hello'}
end.
```

See Also

AscToHeap
DisposeAsc
StrFromHeap (TPSTRING)

Declaration

```
procedure AscLeftPad(var A : Asciiz; Len : word; var B : Asciiz);
```

Purpose

Left-pad the Asciiz in A to length Len with blanks, returning B.

Comments

If the length of A already matches or exceeds Len, then B returns the same as A. Otherwise, A is transferred to B and then blanks are added to the start of B until the length reaches Len.

Example

```
var
  A, B : Asciiz;
begin
  Str2Asc('hello', A);
  AscLeftPad(A, 10, B); {B gets '     hello'}
end.
```

See Also

AscLeftPadCh
AscPad
LeftPad (TPSTRING)

Declaration

```
procedure AscLeftPadCh(var A : Asciiz; Ch : Char; Len : word; var B : Asciiz);
```

Purpose

Left-pad the Asciiz in A to length Len with Ch, returning B.

Comments

If the length of A already matches or exceeds Len, then B returns the same as A. Otherwise, A is transferred to B and then characters Ch are added to the start of B until the length reaches Len.

Example

```
var
  A, B : Asciiz;
begin
  Str2Asc('hello', A);
  AscLeftPadCh(A, '-', 10, B); {B gets '-----hello'}
end.
```

See Also

AscLeftPad
AscPadCh
LeftPadCh (TPSTRING)

AscLoCase

Declaration

```
procedure AscLoCase(var A, B : Asciiz);
```

Purpose

Lowercase the Asciiz in A, returning B.

Comments

Each character of A is set to lowercase and the result is returned in B. Note that AscLoCase handles correctly all alphabetics in the international character set ('A'..'Z', 'a'..'z', #128..#154, #160..#167).

Example

```
var
  A, B : Asciiz;
begin
  Str2Asc('HELLO', A);
  AscLoCase(A, B); {B gets 'hello'}
end.
```

See Also

StLoCase (TPSTRING)

Declaration

```
procedure AscPad(var A : Asciiz; Len : word; var B : Asciiz);
```

Purpose

Right-pad the Asciiz in A to length Len with blanks, returning B.

Comments

If the length of A already exceeds Len, then B is set to A. Otherwise, B is set to A and then blanks are added to the end of B until its length reaches Len.

Example

```
var
  A, B : Asciiz;
begin
  Str2Asc('hello', A);
  AscPad(A, 10, B); {B gets 'hello     '}
end.
```

See Also

AscLeftPad
AscPadCh
PadCh (TPSTRING)

Declaration

```
procedure AscPadCh(var A : Asciiz; Ch : Char; Len : word; var B : Asciiz);
```

Purpose

Right-pad the Asciiz in A to length Len with Ch, returning B.

Comments

If the length of A already exceeds Len, then B is set to A. Otherwise, B is set to A and then characters Ch are added to the end of B until its length reaches Len.

Example

```
var
  A, B : Asciiz;
begin
  Str2Asc('hello', A);
  AscPadCh(A, '-', 10, B); {B gets 'hello-----'}
end.
```

See Also

AscLeftPadCh
AscPad
PadCh (TPSTRING)

AscToHeap

Declaration

```
function AscToHeap(var A : Asciiz) : AsciizPtr;
```

Purpose

Put Asciiz on heap, returning a pointer, nil if insufficient memory.

Comments

Because Asciiz strings can be so long, it is often desirable to store them on the heap rather than in the data segment where they can quickly consume the available 64K bytes. AscToHeap works in conjunction with AscFromHeap and DisposeAsc.

AscToHeap allocates heap space for an Asciiz and returns a pointer to it. AscFromHeap returns the data from the heap to another variable. DisposeAsc deallocates the heap space when it is no longer needed.

If sufficient heap space is not available, AscToHeap returns a nil pointer. Note that AscToHeap allocates only the exact amount of space needed for the Asciiz passed as its parameter. If instead you want to allocate enough space for the longest Asciiz, use the following statement:

```
New(P); {where P is of type AsciizPtr}
```

Example

```
var
  P : AsciizPtr;
  A : Asciiz;
begin
  Str2Asc('hello', A);
  P := AscToHeap(A); {P^ holds 'hello'#0}
end.
```

See Also

AscFromHeap
DisposeAsc
StrFromHeap (TPSTRING)

Declaration

```
procedure AscTrim(var A, B : Asciiz);
```

Purpose

Return an Asciiz with leading and trailing white space removed.

Comments

White space is defined as any ASCII character whose ordinal value is 32 or less, with the exception of #0, which is used as the Asciiz termination character. If the Asciiz A consists entirely of white space, a null string is returned in B. White space internal to A is passed along unchanged.

Example

```
var A, B : Asciiz;
begin
  Str2Asc(' hello world! ', A);
  AscTrim(A, B); {B gets 'hello world!'}
end.
```

See Also

AscTrimLead
AscTrimTrail
Trim (TPSTRING)

Declaration

```
procedure AscTrimLead(var A, B : Asciiz);
```

Purpose

Return an Asciiz with leading white space removed.

Comments

White space is defined as any ASCII character whose ordinal value is 32 or less, with the exception of #0 which is used as the Asciiz termination character. If the Asciiz A consists entirely of white space, a null string is returned in B. White space internal to A is passed along unchanged.

Example

```
var A, B : Asciiz;
begin
  Str2Asc(' hello world! ', A);
  AscTrimLead(A, B); {B gets 'hello world! '}
end.
```

See Also

AscTrim
AscTrimTrail
TrimLead (TPSTRING)

AscTrimTrail / AscUpCase

Declaration

```
procedure AscTrimTrail(var A, B : Asciiz);
```

Purpose

Return an Asciiz with trailing white space removed.

Comments

White space is defined as any ASCII character whose ordinal value is 32 or less, with the exception of #0 which is used as the Asciiz termination character. If the Asciiz A consists entirely of white space, a null string is returned in B. White space internal to A is passed along unchanged.

Example

```
var
  A, B : Asciiz;
begin
  Str2Asc(' hello world! ', A);
  AscTrimTrail(A, B); {B gets ' hello world!'}
end.
```

See Also

AscTrim
AscTrimLead
TrimTrail (TPSTRING)

Declaration

```
procedure AscUpCase(var A, B : Asciiz);
```

Purpose

Uppercase the Asciiz in A, returning B.

Comments

B gets A with lowercase characters converted to uppercase. Note that AscUpCase handles correctly all alphabetics in the international character set ('A'..'Z', 'a'..'z', #128..#154, #160..#167).

Example

```
var
  A, B : Asciiz;
begin
  Str2Asc('hello!', A);
  AscUpCase(A, B); {B gets 'HELLO!'}
end.
```

See Also

AscLoCase
StUpCase (TPSTRING)

Declaration

```
function CompAsc(var A1, A2 : Asciiz) : AscCompareType;
```

Purpose

Return less, equal, greater if A1<A2, A1=A2, or A1>A2.

Comments

Two Asciiz strings are compared using the same rules as for normal strings. The function returns an enumerated type taking the values AscLess, AscEqual, and AscGreater.

Examples

See the entry for CompString (TPSTRING) for examples.

See Also

CompString (TPSTRING) CompUCAsc

Declaration

```
function CompUCAsc(var A1, A2 : Asciiz) : AscCompareType;
```

Purpose

Compare two Asciiz's in a case insensitive manner.

Comments

Two Asciiz strings are compared using the same rules as for normal strings. During comparison, all lowercase characters are converted to uppercase, without affecting the actual parameters. The function returns an enumerated type taking the values AscLess, AscEqual, and AscGreater. Note that CompUCAsc handles correctly all alphabetics in the international character set ('A'..'Z', 'a'..'z', #128..#154, #160..#167).

Examples

See the entry for CompUCString (TPSTRING) for examples.

See Also

CompAsc CompUCString (TPSTRING)

ConcatAsc / ConcatStr

Declaration

```
procedure ConcatAsc(var A, B, C : Asciiz);
```

Purpose

Concatenate two Asciiz strings.

Comments

This procedure is analogous to the normal string expression C := A+B. If the resulting length exceeds the maximum for Asciiz strings, the result is truncated.

Example

```
var A, B, C : Asciiz;
begin
  Str2Asc('hello ', A);
  Str2Asc('world!', B);
  ConcatAsc(A, B, C); {C gets 'hello world!'}
end.
```

See Also

ConcatStr

Declaration

```
procedure ConcatStr(var A : Asciiz; S : string; var C : Asciiz);
```

Purpose

Concatenate a string to an Asciiz.

Comments

This procedure is analogous to the normal string expression C := A+S. In this case a normal Turbo string S is concatenated to the Asciiz A to return the Asciiz C. If the resulting Asciiz exceeds the maximum length, it is truncated.

Example

```
var A, B : Asciiz;
begin
  Str2Asc('hello ', A);
  ConcatStr(A, 'world!', B); {B gets 'hello world!'}
end.
```

See Also

ConcatAsc

Declaration

```
procedure CopyAsc(var A : Asciiz; Start, Len : Word; var O : Asciiz);
```

Purpose

Return a substring of A.

Comments

This procedure is analogous to the Copy function for normal strings. Len characters are taken from position Start of A and returned in O. Note that Start = 0 refers to the first character of A.

If more characters are requested from A than exist, those that exist are returned. If the Start position exceeds the length of A, the null string is returned.

Example

```
var
  A, B : Asciiz;
begin
  Str2Asc('hello ', A);
  CopyAsc(A, 1, 3, B); {B gets 'ell'}
  CopyAsc(A, 4, 10, B); {B gets 'o'}
end.
```

Declaration

```
procedure DeleteAsc(var A : Asciiz; Start, Len : Word);
```

Purpose

Delete Len characters of A, starting at position Start.

Comments

This procedure is analogous to the Delete function for normal strings. Len characters are removed from position Start of A. Note that Start = 0 refers to the first character of A.

If the deletion calls for more characters from A than exist, then the existing ones are deleted. If the Start position exceeds the length of A, then A is not changed.

Example

```
var
  A : Asciiz;
begin
  Str2Asc('hello ', A);
  DeleteAsc(A, 10, 10); {A is not changed}
  DeleteAsc(A, 1, 1); {A gets 'hllo'}
  DeleteAsc(A, 0, 2); {A gets 'lo'}
end.
```

DisposeAsc

Declaration

```
procedure DisposeAsc(P : AsciizPtr);
```

Purpose

Dispose of heap space pointed to by P.

Comments

Because Asciiz strings can be so long, it is often desirable to store them on the heap rather than in the data segment where they can quickly consume the available 64K bytes. DisposeAsc works in conjunction with AscToHeap.

AscToHeap allocates heap space for an Asciiz and returns a pointer to it. DisposeAsc deallocates the heap space when it is no longer needed.

If the pointer P is nil, DisposeAsc does nothing. DisposeAsc should not be used to dispose of pointers allocated directly with a New statement. In those cases, use the built-in Turbo Dispose procedure to deallocate the heap space.

Example

```
var
  P : AsciizPtr;
  A : Asciiz;
begin
  Str2Asc('hello', A);
  P := AscToHeap(A);
  DisposeAsc(P); {6 bytes of heap space freed, and P gets nil}
end.
```

See Also

AscToHeap

DisposeString (TPSTRING)

Declaration

```
procedure InsertAsc(var Obj, A : Asciiz; Start : Word);
```

Purpose

Insert Asciiz Obj at position Start of A.

Comments

This routine is analogous to the Insert procedure for normal strings. The Obj string is inserted at position Start of the Asciiz A. Note that Start = 0 refers to the first position in A.

If inserting Obj causes A to exceed the maximum length for Asciiz strings, A is truncated. If Start is a position beyond the end of A, Obj is concatenated to A.

Example

```
var A, B : Asciiz;
begin
  Str2Asc('hello ', A);
  Str2Asc('world!', B);
  InsertAsc(A, B, 0); {B gets 'hello world!'}
end.
```

See Also

InsertStr

Declaration

```
procedure InsertStr(Obj : string; var A : Asciiz; Start : Word);
```

Purpose

Insert string Obj at position Start of A.

Comments

This routine is analogous to the Insert procedure for normal strings. Normal Turbo string Obj is inserted at position Start of the Asciiz A. Note that Start = 0 refers to the first position in A.

If inserting Obj causes A to exceed the maximum length for Asciiz strings, A is truncated. If Start is a position beyond the end of A, Obj is concatenated to A.

Example

```
var A : Asciiz;
begin
  Str2Asc('world!', A);
  InsertStr('hello ', A, 0); {B gets 'hello world!'}
end.
```

See Also

InsertAsc

LenAsc / PosAsc

Declaration

```
function LenAsc(A : Asciiz) : Word;
```

Purpose

Return the length of an Asciiz string.

Comments

The length of an Asciiz string is the count of all of its characters, not including the trailing null that terminates the string. Thus, the length may range from 0 to MaxAsciiz.

Because LenAsc performs a counting operation, it is implemented as an assembly language inline macro. As such, it consumes 12-14 bytes more code space per call than would an actual call to a function. Moreover, because each scan for the end of a string takes time, LenAsc should be called a minimum number of times by assigning its result to a word type variable.

Examples

```
var A : Asciiz;  L : word;
begin
  Str2Asc('hello', A);
  L := LenAsc(A); {L gets 5}
  DeleteAsc(A,0,5);
  L := LenAsc(A); {L gets 0}
end.
```

Declaration

```
function PosAsc(var ObjA, A : Asciiz) : Word;
```

Purpose

Return the position of ObjA in A.

Comments

The PosAsc routine is analogous to the Pos function for normal Turbo strings. It returns the position of ObjA in A. Note that the first position of A is 0. If ObjA is not found in A, then PosAsc returns $FFFF (which may be referred to using the defined constant NotFound).

Examples

```
var A, B : Asciiz;  Posn : word;
begin
  Str2Asc('hello world!', A);
  Str2Asc('aardvark', B);
  Posn := PosAsc(B, A); {Posn gets $FFFF}
  Str2Asc('hello', B);
  Posn := PosAsc(B, A); {Posn gets 0}
end.
```

See Also

PosStr

Search (TPSTRING)

Declaration

```
function PosStr(Obj : string; var A : Asciiz) : Word;
```

Purpose

Return the position of the string Obj in A, returning NotFound if not found.

Comments

The PosStr routine is analogous to the Pos function for normal Turbo strings. It returns the position of the normal Turbo string Obj in the Asciiz A. Note that the first position of A is 0. If Obj is not found in A, then PosStr returns $FFFF (which may be referred to using the defined constant NotFound).

Examples

```
var
  A : Asciiz;
  Posn : word;
begin
  Str2Asc('hello world!', A);
  Posn := PosStr('aardvark', A); {Posn gets $FFFF}
  Posn := PosStr('hello', A); {Posn gets 0}
end.
```

See Also

PosAsc

Search (TPSTRING)

ReadLnAsc

Declaration

```
function ReadLnAsc(var F : Text; var A : Asciiz) : Boolean;
```

Purpose

Read an Asciiz from text file, returning True if successful.

Comments

This function lets you read long strings from text files. F is a text file previously opened for reading with any buffer size. The parameter A will return an Asciiz string from the file, ending at the next carriage return / line feed combination (ASCII #13#10).

If an I/O error occurs, ReadLnAsc returns False. If F is already at end of file, ReadLnAsc returns False and A is the null string. If the line of text exceeds the maximum length for an Asciiz, the remainder will be truncated.

Example

```
var
  F : text;
  A : Asciiz;
begin
  Assign(F, 'LONG.TXT');
  Reset(F);                                  {open for reading}
  while not eof(F) do begin                  {read while not end of file}
    if not ReadLnAsc(F, A) then              {read the next line}
      halt;                                  {halt on error}
    if not WriteAsc(Output, A) then          {write the Asciiz to screen}
      halt;                                  {halt on error}
    WriteLn;                                 {add end of line}
  end;
end.
```

See Also

WriteAsc

Declaration

```
procedure Str2Asc(S : string; var A : Asciiz);
```

Purpose

Convert a Turbo string into an Asciiz.

Comments

This procedure provides a convenient way to initialize Asciiz strings. It converts a normal Turbo string (up to 255 characters long) into an Asciiz. The procedure may also be handy when interfacing Turbo procedures to various MS-DOS functions that call for Asciiz strings. Be warned that any null character (ASCII #0) embedded in the Turbo string will cause a premature termination of the Asciiz.

Example

```
var A : Asciiz;
begin
  Str2Asc('hello', A); {A gets 'hello'#0}
end.
```

See Also

Asc2Str

Declaration

```
function WriteAsc(var F : Text; var A : Asciiz) : Boolean;
```

Purpose

Write an Asciiz to text file, returning True if successful.

Comments

This routine lets you write long strings to text files. F is any Turbo text file previously opened for writing. The Asciiz A is written to the file without its terminating null. If any I/O error occurs, the function returns False.

Example

```
var F : text;
    A : Asciiz;
begin
  Assign(F, 'LONG.TXT'); Reset(F);     {open for reading}
  while not eof(F) do begin            {read while not end of file}
    if not ReadLnAsc(F, A) then halt; {read the next line}
    WriteAsc(Output, A);               {write the Asciiz to screen}
    WriteLn;                           {write a carriage return}
  end;
end.
```

See Also

ReadLnAsc

D. TPSTRDEV: Formatted Strings

As indicated earlier, TPSTRDEV is a very small unit. It exports only a single global variable (TpStr) and two simple routines (ReadStr and ReturnStr) to the outside world. Its simplicity is a bit deceptive, however. TPSTRDEV takes advantage of one of the powerful features introduced in Turbo Pascal 4.0, the "text file device driver," to allow you to use all the formatting capabilities of the Write statement without sending output to a file or device.

Using TPSTRDEV is very simple: all you have to do is add it to the USES clause at the start of your program. The TPSTRDEV unit will initialize itself automatically at the start of the program, making the TpStr "device" look as if it were part of the language. To create formatted strings with it, you just use a normal Write statement. For example:

```
Write(TpStr, 'Elapsed time: ', Hours:2, ' hours, ', Minutes:2, ' minutes.');
```

You can then retrieve the formatted string in one of two ways. You can call ReadStr to load the string into a variable, or you can call ReturnStr:

```
ReadStr(MyString);
MyString := ReturnStr;
```

ReadStr is more efficient than ReturnStr (in terms of speed and stack usage) when all you need to do is put the string in a variable. ReturnStr is more flexible, since you can use the string it returns as an argument in a procedure or function call.

'ariables

```
TPStr : Text;
```

This is the 'device' that you write to to create formatted strings using TPSTRDEV. For example,

```
Write(TPStr, Pi:0:2);
```

would create a string containing '3.14'.

Declaration

```
procedure ReadStr(var S : string);
```

Purpose

'Read' a string into S from our string buffer.

Comments

ReadStr loads the current contents of the string buffer associated with TPStr into a string variable and empties the buffer. In cases where the result is not needed immediately, it is faster and more efficient than ReturnStr.

Example

```
Real1 := 6.95;
Write(TPStr, 'Rate: $', Real1:0:2, ' per hour.');
ReadStr(MyString);
```

MyString would contain 'Rate: $ 6.95 per hour.' after the call to ReadStr.

See Also

ReturnStr

Declaration

```
function ReturnStr : string;
```

Purpose

Return the contents of our string buffer.

Comments

ReturnStr returns the current contents of the string buffer associated with TPStr and empties the buffer. Unlike ReadStr, it can be used to pass the contents of the string buffer to another procedure or function without using an intermediate string variable.

Example

```
Real1 := 6.95;
Write(TPStr, 'Rate: $', Real1:0:2, ' per hour.');
FastWrite(ReturnStr, 1, 1, TextAttr);
```

FastWrite would be passed a string containing 'Rate: $ 6.95 per hour.'.

See Also

ReadStr

5. DOS/BIOS Calls

A. Overview

Turbo Pascal has long given programmers easy access to DOS and the BIOS in general, but for the most part it has required you to write your own routines to perform specific tasks. The DOS unit, a standard part of the Turbo Pascal libraries, eases the programmer's burden to a considerable degree, but even they leave a large number of gaps to be filled. The TPDOS unit in Turbo Professional aims to fill most of these gaps. Unlike the TPCRT unit, which is a plug-in replacement for Turbo's own CRT unit, TPDOS simply complements the DOS unit. When combined, these two units contain most of the routines you should ever need to harness the power of DOS and the BIOS.

The primary emphasis in TPDOS is on file handling and disk drive access. Among other things you'll find routines to: count the number of installed drives; select a new drive; classify a drive; read and write absolute disk sectors; get and set the current Disk Transfer Address (DTA); turn disk write verification on and off; and test for the existence of a file, either in the current directory or anywhere on DOS's PATH. More surprisingly, you'll also find routines that allow you to do random access on text files. TPDOS contains text file equivalents for Turbo Pascal's Seek, FileSize, FilePos, and Flush routines.

TPDOS also contains a variety of routines that give you access to the DOS 3.x concurrent print facility. Once you've determined that PRINT.COM is installed by calling the PrintInstalled function, you can send a file to the print spooler simply by calling SubmitPrintFile, and you can stop the print job just as easily by calling either CancelPrintFile or CancelAllPrintFiles. You can even examine the list of files waiting to be printed by calling GetPrintStatus.

Perhaps the most unusual routine in TPDOS is ExecDos. Although the Exec routine in Turbo's DOS unit was a welcome addition, it presents you with two problems. First, it requires you in many cases to locate COMMAND.COM by searching through the DOS environment for the current COMSPEC setting; if you don't know how to find COMMAND.COM, you can't "shell to DOS" using Exec. Second, it forces you to determine at compile time the maximum amount of heap space your program might require; if you set the figure high to give yourself plenty of elbow room, you can radically limit the number of things that can be done with an Exec call.

ExecDos deals with both of these problems. When appropriate, it automatically

locates COMMAND.COM for you. All you have to do is pass it the name of the DOS command to be executed, and a boolean flag indicating whether or not you've given it the complete pathname of an executable file. More importantly, ExecDos allows you to allocate all available memory for your program without sacrifice. Before executing your DOS command, ExecDos moves Turbo's free list down in memory and returns all memory above the current top of the heap to DOS for its own use. After the command has been executed, all the memory given to DOS is reallocated and the free list is restored to its previous position, so that your program will once again have access to all available memory.

ExecDos has one other unusual feature, in that its third parameter is the address of a routine (written by you) that will save and restore the screen and display status messages for the user. It thus gives you all the flexibility of a routine written especially for a particular program, without sacrificing the convenience of a conventional library routine.

Note that most of the other units in the Turbo Professional library also contain routines for accessing DOS and the BIOS. If you can't find what you need in TPDOS, chances are good that you can find it elsewhere. TPCRT contains a wide variety of routines for manipulating the cursor, writing to and reading from video memory, and reading the keyboard. TPINT and TPTSR contain routines for working with interrupt vectors and making a program terminate and stay resident. TPSTRING contains routines for working with DOS file names. And so on. For a complete list of the procedures and functions in the Turbo Professional library, see the index of Procedures and Functions at the end of each volume of the manual.

The TPENHKBD unit is grouped with the TPDOS unit because its main function is to interact with the BIOS. Unlike TPDOS, however, TPENHBD does not provide an extensive list of procedures and functions for you to call. Instead, it invisibly installs a pair of interrupt handlers that trap all system calls that request keyboard input. Their purpose is to do two things: to make it easy for you to write programs that support enhanced keyboards (the newer ones with the 12 function keys across the top), and to allow older keyboards to emulate the enhanced keyboards wherever possible.

B. TPDOS: Assorted DOS/BIOS Calls

This section describes all of the types, variables, constants, procedures and functions found in the interface section of the TPDOS unit.

Types

```
ActionCodeType =
  (ExecSaveScreen, ExecShowMemory, ExecPauseAfterRun, ExecRestoreScreen);
```

Codes passed by ExecDos to a UserRoutine. The ExecSaveScreen code is passed first, at an opportune time for saving the screen (using SaveWindow, for example). Note that the screen buffer to be used *must* have been allocated before the call to ExecDos. The ExecShowMemory code is passed just before DOS is called. At the same time, a second parameter (a Word) is passed, indicating the approximate amount of memory available (in kilobytes). The ExecPauseAfterRun code is passed when the Exec call returns. You may want to use this opportunity to ask the user to "Press any key...", to avoid wiping out the contents of the screen before the user can see them. The ExecRestoreScreen code is passed immediately after that, giving you a chance to restore the screen.

```
DiskClass = (Floppy360, Floppy720, Floppy12, Floppy144, OtherFloppy, Bernoulli,
             HardDisk, RamDisk, SubstDrive, UnknownDisk, InvalidDrive,
             NovellDrive, CDRomDisk);
```

This enumerated type defines the disk types that can be identified by GetDiskClass, as well as several types indicating that a particular error occurred.

Variables

```
IoResultPtr : Pointer;
```

If not nil (the default value), this must point to a routine like Int24Result that returns an I/O error code of type Word.

Constants

```
MinSpaceForDos : Word = 20000;
```

Minimum number of bytes for DOS shell to run. If ExecDos cannot free up this much space for DOS before the call, it will abort.

```
StackSafetyMargin : Word = 1000;
```

Used by ExecDos when moving the free list downward in memory. If SizeOf(free list) + StackSafetyMargin <= SP (current stack pointer), the free list will be stored at the bottom of the stack, otherwise on the heap.

```
StdInHandle = 0;
StdOutHandle = 1;
StdErrHandle = 2;
StdPrnHandle = 4;
```

Standard DOS file handles (standard input, standard output, standard error output, standard printer).

CancelAllPrintFiles / CancelPrintFile

Declaration

```
procedure CancelAllPrintFiles;
```

Purpose

Cancels all files in the print queue.

Comments

All printing is stopped immediately. PRINT will not finish printing the current file.

See Also

CancelPrintFile

Declaration

```
procedure CancelPrintFile(FileMask : string);
```

Purpose

Cancels the files matched by the file mask passed in FileMask.

Comments

This routine lets you remove individual files from the print queue. FileMask is any valid DOS file mask, and may include wildcard characters ('?' and '*'). If the file currently being printed is matched by the mask, DOS will stop printing it immediately and go on to the next file in the queue.

Examples

```
CancelPrintFile('*.DOC');
```

Cancel the printing of all files with an extension of 'DOC'.

```
var
  CurrentPrintFile : String[64];
begin
  Write('File to print: ');
  ReadLn(CurrentPrintFile);
  if SubmitPrintFile(CurrentPrintFile) <> 0 then {error};
  {-- something --}
  CancelPrintFile(CurrentPrintFile);
end;
```

Cancel the file currently being printed.

See Also

CancelAllPrintFiles

Declaration

```
function DefaultDrive : Char;
```

Purpose

Returns the default drive as an uppercase letter.

Comments

Returns a character in the range 'A'..'Z'. To convert the result to a drive number, you'd say

```
DriveNumber := Ord(DefaultDrive)-64;
```

Example

```
WriteLn('Current drive: ', DefaultDrive);
```

Would display 'Current drive: C' if C: is the current drive.

See Also

NumberOfDrives SelectDrive

Declaration

```
function DOSVersion : Word;
```

Purpose

Returns the DOS version number.

Comments

The result is in BCD (Binary Coded Decimal) format, with the major version number in the high byte, and the minor version number in the low byte. For example, under DOS 3.1, the result would be $030A ($0A = 10); under DOS 2.11, $020B ($0B = 11); under DOS 3.3, $031E ($1E = 30); and so on.

Note: In Turbo Pascal 5.0 and higher, Borland added a function called DosVersion to the DOS unit. Unfortunately, it returns the major version in the low byte and the minor version in the high byte--just the opposite of what this routine returns. You would therefore be well-advised to use a "unit override" when calling either routine (TpDos.DosVersion or Dos.DosVersion).

Examples

```
if TpDos.DOSVersion < $0300 then
  WriteLn('This program requires DOS 3.0 or higher');
```

Test for DOS 3.x or higher.

```
if TpDos.DOSVersion >= $0400 then
  WriteLn('Warning: Program not tested with DOS 4.00 or higher');
```

Test for DOS 4.00 or higher.

EndPrintStatus

Declaration

```
procedure EndPrintStatus;
```

Purpose

Releases the spooler from the GetPrintStatus procedure.

Comments

Reenables the print spooler after a call to GetPrintStatus, so that printing can resume.

See Also

GetPrintStatus

Declaration

```
function ExecDos(Command : string; UseSecond : Boolean;
                 UserRoutine : Pointer) : Integer;
```

Purpose

Execute any DOS command.

Comments

ExecDos, which acts as a front end for Turbo Pascal's Exec procedure, compresses the heap temporarily in order to free up as much memory as possible for the program being executed. The heap is restored to its original condition after the Exec call.

ExecDos also tries to relieve you of the need to restore interrupt vectors before executing another program. When compiled with Turbo Pascal 4.0, it automatically restores interrupts $23 (DOS control break) and $24 (DOS critical error) before making the Exec call, and sets them back to their previous value afterward. When compiled with Turbo Pascal 5.0 or higher, it uses the SwapVectors routine in the DOS unit to temporarily restore all vectors grabbed by the SYSTEM unit.

Command is the name of a program to be executed or an empty string (which means "shell to DOS"). UseSecond determines whether a second copy of COMMAND.COM is loaded to process the Exec call. If it is False, Command *must* be the full pathname of the program to be executed. (See the entry for the ExistOnPath function.) UserRoutine points to a user-written routine that will be called to display status information, etc. It should be of the form

```
procedure UserRoutine(ActionCode : ActionCodeType; Param : Word);
```

and it *must* be a global procedure compiled under the FAR model (by using the {$F+} compiler directive). For a discussion of action codes, see the entry for ActionCodeType. If you don't want or need this capability, pass a nil pointer instead.

ExecDos return codes are as follows:

0	Success
-1	Insufficient memory to store Turbo's free list
-2	DOS setblock error before EXEC call
-3	DOS setblock error after EXEC call (critical error)
-4	Insufficient memory to run DOS command
else	a DOS error code

Warning: You may not use ExecDos to execute a TSR. ExecDos gives away part of your programs memory allocation on the assumption that it will be able to recover it for you. If it is unable to do so, as it would be if you tried to EXEC a TSR, it will return with an error code (probably of -3).

Examples

```
Code := ExecDos('C:\TURBO\TPC.EXE '+ProgName, False, nil);
```

Call Turbo to compile ProgName. Since the UseSecond parameter is False, the complete pathname for TPC.EXE must be specified.

```
Code := ExecDos('MAKEMENU', True, nil);
```

Execute MAKEMENU. A second copy of COMMAND.COM will be loaded to find and execute the program.

```
var
  P : Pointer;
  BufSize : Word;
  Code : Integer;

{$F+}
procedure MyUserRoutine(ActionCode : ActionCodeType; Param : Word);
var
  Key : Word;
begin
  case ActionCode of
    ExecSaveScreen :    {don't allocate screen buffer, save screen}
      if not SaveWindow(1, 1, ScreenWidth, ScreenHeight, False, P) then
        {can't fail--buffer already allocated};
    ExecShowMemory :    {show memory available for shell}
      WriteLn('Approximately ', Param, 'K bytes available');
    ExecPauseAfterRun : {make sure screen isn't cleared too fast}
      begin
        Write('Press any key to continue...');
        Key := ReadKeyWord;
      end;
    ExecRestoreScreen :  {restore screen, don't dispose of screen buffer}
      RestoreWindow(1, 1, ScreenWidth, ScreenHeight, False, P);
  end;
end;
{$F-}
...
{allocate a screen buffer}
BufSize := 2*ScreenWidth*ScreenHeight;
GetMem(P, BufSize);
Code := ExecDos('', True, @MyUserRoutine);
FreeMem(P, BufSize);
```

Shell to DOS. MyUserRoutine is a user-written action code handler.

See Also

Declarations
ExistOnPath
MinSpaceForDos
StackSafetyMargin

Declaration

```
function ExistFile(FName : String) : Boolean;
```

Purpose

Return True if FName is found.

Comments

Because ExistFile uses the standard Turbo Reset routine, it will not ordinarily find files that are marked read-only. If this is a problem, you can use a "shell" routine that resets Turbo's FileMode variable (default value is 2) before calling ExistFile, like so:

```
function ExistAnyFile(FName : String) : Boolean;
var
  SaveMode : Byte;
begin
  SaveMode := FileMode;        {save current setting}
  FileMode := 0;               {read access only}
  ExistAnyFile := ExistFile(FName);
  FileMode := SaveMode;        {restore previous setting}
end;
```

If you are running on a network, you may want to use a value other than 0 or 2.

ExistFile does not accept wildcard characters, and it looks only in the current directory or in the directory specified in FName.

Examples

```
Found := ExistFile('*.DOC');
```

Incorrect! ExistFile does not accept wildcard characters.

```
Found := ExistFile('C:\AUTOEXEC.BAT');
```

ExistFile would look for 'AUTOEXEC.BAT' in 'C:\'.

```
Found := ExistFile('AUTOEXEC.BAT');
```

ExistFile would look in the current directory for 'AUTOEXEC.BAT'.

See Also

ExistOnPath

ExistOnPath

Declaration

```
function ExistOnPath(FName : String; var FullName : String) : Boolean;
```

Purpose

Return True if FName is found.

Comments

Unlike ExistFile, this routine can often locate files that are not in the current directory. If FName contains a drive and/or path identifer, only the specified drive and directory is searched. But if only a filename is specified, ExistOnPath will search, in order:

- the current directory
- the executable program's directory (under DOS 3.0 or higher only)
- all directories on DOS's PATH

It is, therefore, particularly useful for locating any data files, configuration files, and so on that might be required by your program.

ExistOnPath does not return a full pathname if the file is found in the current directory or if the FName parameter specifies a directory. If you need a full pathname, call TPSTRING.FullPathname after calling ExistOnPath.

Like ExistFile, ExistOnPath does not accept wildcards, and it cannot locate files marked read-only. The solution to the problem is the same as for ExistFile--use a shell routine, like so:

```
function ExistAnywhere(FName : String; var FullName : String) : Boolean;
var
  SaveMode : Byte;
begin
  SaveMode := FileMode;        {save current setting}
  FileMode := 0;               {read access only}
  ExistAnywhere := ExistOnPath(FName, FullName);
  FileMode := SaveMode;        {restore previous setting}
end;
```

Examples

```
Found := ExistOnPath('*.DOC', FullName);
```

Incorrect! ExistOnPath does not accept wildcard characters.

```
Found := ExistOnPath('C:\AUTOEXEC.BAT', FullName);
```

ExistOnPath would look for 'AUTOEXEC.BAT' only in 'C:\'.

```
Found := ExistOnPath('AUTOEXEC.BAT', FullName);
```

ExistOnPath would do a full search for 'AUTOEXEC.BAT'.

See Also

ExistFile

FileHandlesLeft / FileHandlesOpen

Declaration

```
function FileHandlesLeft : Byte;
```

Purpose

Return the number of available file handles.

Comments

This routine simply attempts to open up to 255 files, and returns the number that it could successfully open. All files that were opened are closed on exit. This routine is intended primarily for purposes of debugging.

Example

```
{$IFDEF Debug}
WriteLn('There are ', FileHandlesLeft, ' file handles available.');
{$ENDIF}
```

A typical use for FileHandlesLeft, a debugging statement that displays the number of file handles currently available.

Declaration

```
function FileHandlesOpen(CountDevices : Boolean) : Byte;
```

Purpose

Return the number of open files owned by a program.

Comments

This routine returns the number of files that your program currently has open. The CountDevices parameter indicates whether or not the standard DOS devices (those with handles in the range StdInHandle..StdPrnHandle) should be included in the count. The number returned will always be in the range 0..255 if CountDevices is True, or 0..249 if it is False. Keep in mind that there is no guarantee that all 255 file handles will be available for use by your program unless "FILES=255" appears in the user's CONFIG.SYS file and you use the capabilities provided in EXTEND.LZH on the BONUS disk. The default setting for FILES is only 8, unfortunately. This routine is intended primarily for purposes of debugging.

Example

```
{$IFDEF Debug}
WriteLn('There are now ', FileHandlesOpen(True), ' files open.');
{$ENDIF}
```

A typical use for FileHandlesOpen, a debugging statement that displays the number of files that are currently open.

GetDiskClass

Declaration

```
function GetDiskClass(Drive : Char; var SubstDriveChar : Char) : DiskClass;
```

Purpose

Return the disk class for the drive with the specified letter.

Comments

Given a drive letter ('A'..'Z'), this routine returns a value indicating what kind of disk is installed in the specified drive. The possible return values are:

```
Floppy360      A 360K, 5 1/4 floppy disk
Floppy720      A 720K, 3 1/2 floppy disk
Floppy12       A 1.2 meg, 5 1/4 floppy disk
Floppy144      A 1.44 meg, 3 1/2 floppy disk
OtherFloppy    Another kind of floppy disk (e.g. a single-sided disk)
Bernoulli      A Bernoulli drive
HardDisk       Any hard disk
RamDisk        A RAM disk created with VDISK.SYS,RAMDRIVE.SYS, etc.
SubstDrive     A logical drive created by SUBST or ASSIGN
UnknownDisk    A drive that could not be classified
InvalidDrive   Indicates that the drive does not exist or isn't ready
NovellDrive    A logical drive located on a Novell server
CDRomDisk      A CD-ROM drive
```

Limitations: GetDiskClass cannot recognize the type of a floppy drive, only the type of the disk (if any) that is in it. For example, a 1.2 meg floppy drive is classified as Floppy360 if it has a 360KB disk in it. Also, GetDiskClass does not identify several rarely used floppy disk types (single-sided floppies, for example).

The return value SubstDrive is used for two special cases. The first case is when the DOS ASSIGN program is used to treat one drive as another. For example,

```
ASSIGN B=C
```

causes all disk I/O for drive B to be routed to drive C. When GetDiskClass recognizes this situation, it sets the return value to SubstDrive and sets SubstDriveChar to the letter of the actual drive being used. In this example, SubstDriveChar is set to C.

The second case in which the return value is SubstDrive is when the DOS SUBST program is used to create a new logical drive from an existing drive or directory. For example,

```
SUBST H: C:\TURBO
```

allows you to refer to the directory "C:\TURBO" as drive H. When GetDiskClass recognizes this situation, it sets the return value to SubstDrive and sets SubstDriveChar to the letter of the actual drive being used. In this example, SubstDriveChar is set to C. If the return value is anything other than SubstDrive, SubstDriveChar contains the same drive letter as Drive.

Declaration

```
function GetDiskInfo(Drive : Byte; var ClustersAvailable, TotalClusters,
                     BytesPerSector, SectorsPerCluster: Word) : Boolean;
```

Purpose

Return technical info about the specified drive.

Comments

This routine returns information about the organization of a given drive. You can use it to calculate the total capacity of a disk, the number of bytes free, or to obtain data to be used in conjunction with ReadDiskSectors and/or WriteDiskSectors.

Drive is a number indicating the drive of interest: 0 = default, 1 = A, 2 = B, and so on. To convert a drive letter to a drive number, use the following formula:

```
DriveNumber := Ord(UpCase(DriveLetter))-64;
```

The function returns False if the drive number is invalid. Otherwise, it returns the following values: ClustersAvailable (number of unused clusters on the disk), TotalClusters (total number of clusters on the disk), BytesPerSector (number of bytes in a given disk sector, almost always 512), and SectorsPerCluster (number of sectors in a cluster, usually 1, 2, or 4).

Formulas:

```
BytesPerCluster := LongInt(SectorsPerCluster) * LongInt(BytesPerSector);
DiskCapacity := LongInt(TotalClusters) * BytesPerCluster;
FreeSpace := LongInt(ClustersAvailable) * BytesPerCluster;
```

Example

```
var
  BytesPerCluster, DiskCapacity, FreeSpace : LongInt;
  ClustersAvailable, TotalClusters,
  BytesPerSector, SectorsPerCluster: Word;
begin
  if GetDiskInfo(0, ClustersAvailable, TotalClusters, BytesPerSector,
    SectorsPerCluster) then begin
      BytesPerCluster := LongInt(SectorsPerCluster) * LongInt(BytesPerSector);
      DiskCapacity := LongInt(TotalClusters) * BytesPerCluster;
      FreeSpace := LongInt(ClustersAvailable) * BytesPerCluster;
      WriteLn('Info about drive ', DefaultDrive, ':');
      WriteLn('Bytes per sector: ', BytesPerSector);
      WriteLn('Bytes per cluster: ', BytesPerCluster);
      WriteLn('Disk capacity: ', DiskCapacity);
      WriteLn('Bytes available: ', FreeSpace);
    end;
end;
```

See Also

ReadDiskSectors

WriteDiskSectors

GetDta

Declaration

```
procedure GetDta(var DTAptr : pointer);
```

Purpose

Return the DOS DTA pointer.

Comments

GetDta returns a pointer to the currently active DOS DTA. This routine is important when the DTA is to be temporarily redirected with SetDta, then restored to the original one. This is especially important for memory resident programs that must access the disk via DOS. However, you won't need to call GetDta/SetDta in this case because the Turbo Professional popup routines do it for you automatically.

Example

```
var
  OldDta : pointer;
  SR : SearchRec;
begin
  GetDta(OldDta); {Save current DTA}
  SetDta(@SR);    {Point DTA to a SearchRec mask}
  {--Perform file or directory operations--}
  SetDta(OldDta); {Restore the original DTA}
end;
```

See Also

SetDta

Declaration

```
function GetEnvironmentString(SearchString : string) : string;
```

Purpose

Return a string from the environment.

Comments

This routine allows you to locate strings assigned to environment variables, such as 'COMSPEC', 'PATH', and 'PROMPT'. The search is case-insensitive, and you do not need to specify '=' at the end of the search string.

Examples

```
WriteLn(GetEnvironmentString('COMS'));
```

Correct *only* if the complete variable name being sought is 'COMS'. This would *not* locate 'COMSPEC'.

```
WriteLn(GetEnvironmentString(' COMSPEC '));
```

Incorrect! The spaces at the ends of the string must be eliminated.

```
WriteLn(GetEnvironmentString('comspec='));
WriteLn(GetEnvironmentString('comspec'));
WriteLn(GetEnvironmentString('COMSPEC='));
WriteLn(GetEnvironmentString('COMSPEC'));
```

All OK, produces the same result--e.g., 'C:\COMMAND.COM'.

GetFileMode

Declaration

```
function GetFileMode(FName : string; var Attr : Word) : Byte;
```

Purpose

Get the attribute of a file.

Comments

This routine simply puts a "shell" around Turbo's GetFAttr routine to make it easier to use. Returns FName's attribute in Attr, and the DOS error code (DosError) as the function result. The attribute is bit-coded as follows:

Bit	Val	Meaning
0	$01	Read-only
1	$02	Hidden
2	$04	System
3	$08	Volume label
4	$10	Subdirectory
5	$20	Archive bit

To see if a file is hidden, for example, you would use the following test:

```
Hidden := (Attr and $02) <> 0;
```

Example

```
var
  FName : string;
  Attr : Word;
begin
  Write('Enter file name:' );
  ReadLn(FName);
  if GetFileMode(FName, Attr) <> 0 then
    WriteLn('File not found')
  else begin
    WriteLn(FName, ' is a ');
    if (Attr and $10) <> 0 then
      WriteLn('directory')
    else
      WriteLn('file');
  end
end;
```

Declaration

```
function GetPrintStatus(var QPtr : Pointer) : Byte;
```

Purpose

Halts printing, returns current error status, puts pointer to the filename queue in the QPtr variable.

Comments

This routine returns a pointer to a series of ASCIIZ strings representing the names of files in the print queue. Each entry is 64 bytes in size, and the end of the queue is marked by a name starting with a null. Printing is suspended from the time you call GetPrintStatus until you issue a call to EndPrintStatus. Note that you can easily convert ASCIIZ strings to normal strings using the Asc2Str routine in TPASCIIZ. GetPrintStatus returns 0 if successful.

Example

```
var
  P : Pointer;
  AP : AsciizPtr absolute P;
  POffset : Word absolute P;
begin
  if GetPrintStatus(P) = 0 then
    while Char(P^) <> #0 do begin {check for null}
      WriteLn(Asc2Str(AP^));      {display current entry}
      Inc(POffset, 64);           {point to next entry}
    end;
  EndPrintStatus;                 {release print spooler}
end;
```

See Also

Asc2Str (TPASCIIZ)

EndPrintStatus

HandleIsConsole / NumberOfDrives

Declaration

```
function HandleIsConsole(Handle : Word) : Boolean;
```

Purpose

Return True if handle is the console device.

Comments

HandleIsConsole allows you, among other things, to determine whether or not a program's I/O has been redirected. This information can be crucial when writing a *filter*, such as BSORT.PAS, which needs to know if the user has redirected input or output as required.

Examples

```
if HandleIsConsole(StdInHandle) then
  WriteLn('Input must be redirected from a file');
if HandleIsConsole(StdOutHandle) then
  WriteLn('Output should be redirected to a file');
```

For another example, see the SortStr routine in BSORT.PAS.

Declaration

```
function NumberOfDrives : Byte;
```

Purpose

Returns the number of logical drives.

Comments

Under DOS 3.0 or greater, the value returned will always be at least 5, because DOS 3.0 and higher supports SUBSTituted drives. Under DOS 2.x, the actual number of drives will be returned.

Example

```
WriteLn('Number of drives, according to DOS: ', NumberOfDrives);
```

Declaration

```
function OpenStdDev(var F : Text; StdHandle : Word) : Boolean;
```

Purpose

Assign the text file to a standard DOS device: 0, 1, 2, or 4.

Comments

This routine lets you easily access the standard input, standard output, standard error output, and standard printer devices. F is any unopened text file variable, and StdHandle is 0 (StdInHandle), 1 (StdOutHandle), 2 (StdErrHandle), or 4 (StdPrnHandle). Among other things, OpenStdDev allows you to write to the standard output device in programs that use CRT or TPCRT. It returns False only if StdHandle is an invalid number.

Example

```
var StdOut : Text;
begin
  if not OpenStdDev(StdOut, StdOutHandle) then {error};
end;
```

Declaration

```
function ParsePath(var InputPath, SearchPath, LeadInPath : string) : Boolean;
```

Purpose

Takes a user entered path, trims blanks, and returns a valid global search path and a valid lead-in path.

Comments

This routine complements the FindFirst and FindNext routines in the standard DOS unit. ParsePath parses a file mask (InputPath) exactly as the DOS COPY command does, and returns a string to be passed on to FindFirst (SearchPath) and another string containing just the path being searched. Leading and trailing blanks are removed, and all characters are converted to upper case. For example, given the string ' \dos ', ParsePath would return '\DOS*.*' in SearchPath and '\DOS\' in LeadInPath. It returns False if InputPath cannot be parsed.

Examples

Given an InputPath of 'c:' ParsePath would return

```
'C:*.*' {SearchPath}
'C:'    {LeadInPath}
```

Given an InputPath of '*.COM' ParsePath would return

```
'*.COM' {SearchPath}
''      {LeadInPath}
```

PrintInstalled

Declaration

```
function PrintInstalled : Boolean;
```

Purpose

Returns True if PRINT.COM is installed.

Comments

All of the routines for accessing DOS's print spooler (PRINT.COM) require DOS 3.0 or higher. If the program is being run under an earlier version, PrintInstalled will always return False. Otherwise, it will return False only if the print spooler is not resident.

Example

```
if not PrintInstalled then
  WriteLn('PRINT.COM not resident');
```

See Also

CancelAllPrintFiles
CancelPrintFile
EndPrintStatus
GetPrintStatus
SubmitPrintFile

Declaration

```
function ReadDiskSectors(Drive : Word; FirstSect : Longint;
                         NumSects : Word; var Buf) : Boolean;
```

Purpose

Read absolute disk sectors.

Comments

This routine provides you access to individual disk sectors, allowing you to inspect even unused portions of a disk. Drive is a number between 0 (= A:) to 25 (= Z:). To convert a drive letter to a drive number, use the following formula:

```
DriveNumber := Ord(UpCase(DriveLetter))-Ord('A');
```

FirstSect is the sector to read from (first sector on a disk is 0). This value should normally be in the range 0..65535, but if DOS 3.31 or higher is in use, the number can be larger. If DOS 3.3 or lower is in use, the high word of FirstSect is ignored.

NumSects is the number of sectors to read. Buf is the buffer to read data into; it *must* be at least NumSects * BytesPerSector bytes (see GetDiskInfo). If ReadDiskSectors fails, it returns False and puts an error code in DosError, interpreted as follows:

High byte (error code generated by BIOS):

128	Time out: drive failed to respond
64	Seek operation failed
32	Controller failed
16	Bad CRC (Cyclical Redundancy Check) – parity error
8	DMA (Direct Memory Access) failure
4	Sector not found
3	Attempt to write on write protected diskette
2	Bad address mark: sector ID marking invalid or not found

Low byte (error code generated by DOS):

12	General failure
11	Read error
10	Write error
8	Sector not found
7	Unknown media; disk format not recognized
6	Seek error
4	Bad CRC
2	Drive not ready
1	Unknown unit: invalid drive number
0	Attempt to write on write protected diskette

ReadDiskSectors is not supported in DOS protected mode due to problems in some DPMI server software.

ReadDiskSectors

Example

```
var Drive : Byte;
    Buffer : array[1..2048] of Byte;
begin
  Drive := Ord(DefaultDrive)-Ord('A');
  if not ReadDiskSectors(Drive, 0, 1, Buffer) then begin
    WriteLn('Error ', DosError, ' reading from ', DefaultDrive);
    DosError := 0;
    Exit;
  end;
end;
```

See Also

GetDiskInfo

WriteDiskSectors

Declaration

```
procedure SelectDrive(Drive : Char);
```

Purpose

Selects the specified drive as default if possible.

Comments

Drive should be a letter in the range 'a'..'z' or 'A'..'Z'. If the drive is invalid, SelectDrive does nothing.

Examples

```
SelectDrive('A');
SelectDrive('b');
SelectDrive('C');
```

See Also

DefaultDrive NumberOfDrives

Declaration

```
function SetBlock(var Paragraphs : Word) : Boolean;
```

Purpose

Change size of DOS memory block allocated to this program.

Comments

This routine, intended primarily for internal use by ExecDos, allows you to increase or decrease the amount of memory allocated to your program. Paragraphs is the new size of the memory block requested. If the request cannot be filled, SetBlock returns False and Paragraphs contains the number of available paragraphs.

Very important: If you change a program's memory block, you must also be certain that Turbo's free list is set up properly. The ExecDos routine in TPDOS.PAS demonstrates how to safely alter the memory block.

Examples

See the ExecDos routine in TPDOS.PAS for examples.

See Also

ExecDos

SetDta

Declaration

```
procedure SetDta(DTAptr : Pointer);
```

Purpose

Set the DOS DTA to point to DTAptr.

Comments

DOS uses a "Disk Transfer Area" for a number of disk operations, notably for directory scanning. A common technique is to point the DOS DTA to a region overlayed by a Pascal record. After DOS writes directory information to this region, the fields of the Pascal record provide an easy means of parsing the data.

In fact, this is the technique used by the FindFirst routine in the DOS unit. FindFirst and FindNext offer an easy way to read the DOS directory, but they do not work well for programs such as TFIND that search directories recursively.

SetDTA calls DOS function $1A. Generally the parameter should be a pointer to a SearchRec (defined in the DOS unit). See TFIND.PAS for a good example.

Example

```
var SR : SearchRec;
begin
  SetDta(@SR); {Points the DTA to SR}
end;
```

See Also

GetDta

Declaration

```
procedure SetRawMode(var F : Text; On : Boolean);
```

Purpose

Set "raw" mode on or off for the specified text file (must be a device).

Comments

This routine allows you to change the mode of a character device (standard input, standard output, standard printer) from "cooked" to "raw" and back again. In cooked mode, DOS expands tabs and interprets control characters such as ^S and ^C when reading from or writing to a character device. In raw mode, all characters are treated as data.

The F parameter passed to SetRawMode must be an open text file associated with a character device. If F is not a device, it does nothing. I/O errors are reported in DOS.DosError.

Probably the most common use for this routine is to put the standard printer device into raw mode. This is often necessary in programs that need to pass control character sequences to the printer. Another common use for SetRawMode is to speed up I/O operations involving the standard input or standard output devices.

Example

```
var
  F : Text;
begin
  Assign(F, 'LPT1);
  Rewrite(F);
  SetRawMode(F, True);
  ...
  Close(F);
  SetRawMode(F, False);
end.
```

Put the printer device in raw mode at beginning of program, and restore cooked mode at the end. In most cases, an exit handler should also be installed that will restore cooked mode even if the program ends prematurely.

SetVerify

Declaration

```
procedure SetVerify(On : Boolean);
```

Purpose

Turn disk write verification on/off.

Comments

SetVerify is most useful in programs that manipulate highly sensitive data. When VERIFY is ON, DOS automatically checks the integrity of all data written to disk.

Examples

```
if not VerifyOn then SetVerify(True);
```

Make sure VERIFY is ON.

```
if UserWantsVerifyOff then SetVerify(False);
```

Turn VERIFY OFF for faster disk writes.

See Also

VerifyOn

Declaration

```
function SubmitPrintFile(FileName : string) : Byte;
```

Purpose

This procedure submits a file to PRINT.COM.

Comments

This routine lets you submit files for concurrent printing under DOS 3.0 and higher, provided that PRINT.COM is installed. Before calling it, you should be certain that both of these conditions are met by calling PrintInstalled.

If the print request can be filled, SubmitPrintFile returns 0; otherwise, one of the following error codes:

1	Invalid function
2	File not found
3	Path not found
4	Too many files open
5	Access denied
8	Print queue full
9	Busy
12	Name too long
15	Invalid drive

Example

```
if not PrintInstalled then
  WriteLn('PRINT.COM not installed')
else
  if SubmitPrintFile('\AUTOEXEC.BAT') <> 0 then
    WriteLn('Cannot print file');
```

See Also

PrintInstalled

TextFileSize / TextFlush

Declaration

```
function TextFileSize(var F : Text) : LongInt;
```

Purpose

Return the size of text file F.

Comments

This routine allows you to determine the current size of any text file opened for input or output. The current contents of the text buffer are accounted for in the result. TextFileSize returns -1 in case of I/O error.

Example

```
var MyFileSize : LongInt;
  F : Text;
begin
  {--open F--}
  MyFileSize := TextFileSize(F);
  if MyFileSize = -1 then WriteLn('Error')
  else WriteLn('Size = ', MyFileSize);
end;
```

Declaration

```
function TextFlush(var F : Text) : Boolean;
```

Purpose

Flush the buffer(s) for a text file.

Comments

This routine empties a text file's buffer. If the file has been opened for output, the current contents of the buffer are written to disk, and DOS's file buffers are flushed using DOS's Duplicate File Handle ($45) and Close File ($3E) functions. If it has been opened for input, the DOS file pointer is moved back to point to the next character to be read (i.e., the location it would be at if there were no buffering). TextFlush returns False in case of I/O error.

Example

```
var F : Text;
begin
  {--open F--}
  if not TextFlush(F) then WriteLn('Error flushing F')
  else WriteLn('F flushed');
end;
```

Declaration

```
function TextPos(var F : Text) : LongInt;
```

Purpose

Return the current position of the logical file pointer.

Comments

TextPos obtains the current position of the physical file pointer, then adjusts that value to account for the current contents of the text file's buffer. The result is the current position of the *logical file pointer*--the position that the physical file pointer would be at if there were no buffering. TextPos returns -1 in case of I/O error. See the entry for TextSeek, below, for an example of how to use TextPos.

See Also

TextSeek

Declaration

```
function TextSeek(var F : Text; Target : LongInt) : Boolean;
```

Purpose

Do a Seek for a text file opened for input.

Comments

This routine allows you to perform random access on text files opened for input. TextSeek returns False if F is not opened for input or if an I/O error occurs. Note that TextSeek sometimes has to empty the text file's buffer, so for performance reasons it is unwise to use it on files with large buffers created with SetTextBuf. TextSeek will, however, avoid doing so whenever possible: if the data being sought is already in the buffer, it will simply reposition the file's buffer pointer, rather than performing a a Seek into the physical file.

Note that Target specifies a byte offset into F, *not* a line number.

Example

```
var SeekDest : LongInt;
    F : Text;
begin
  {--open F--}
  Write('Enter seek destination: ');
  ReadLn(SeekDest);
  if not TextSeek(F, SeekDest) then
    WriteLn('Error')
  else
    WriteLn('SeekDest: ', SeekDest, ' TextPos: ', TextPos(F));
end;
```

See BSORT.PAS for a better example of how TextSeek and TextPos can be used.

See Also

TextPos

TimeMs / VerifyOn

Declaration

```
function TimeMs : LongInt;
```

Purpose

Return time of day in milliseconds since midnight.

Comments

This routine provides a convenient means of measuring a program's run time (in thousandths of a second).

Please note that, although the value returned is in milliseconds, the accuracy of the result is limited by the system timer to about 55 milliseconds (1/18th of a second). If you need to time events more accurately, you should use the TPTIMER unit, found in the BONUS subdirectory on the SRC3 disk.

Example

```
var
  Start, Stop : LongInt;
begin
  Start := TimeMs;
  {-- main program code --}
  Stop := TimeMs;
  WriteLn('Elapsed time: ', (Stop-Start) / 1000.0);
end;
```

Declaration

```
function VerifyOn : Boolean;
```

Purpose

Returns True if disk write verification is on.

Comments

Allows you to determine if DOS is verifying the integrity of data written to disk.

Example

```
Write('VERIFY is ');
if VerifyOn then
  WriteLn('ON')
else
  WriteLn('OFF');
```

See Also

SetVerify

Declaration

```
function WriteDiskSectors(Drive : Word; FirstSect : Longint;
                          NumSects : Word; var Buf) : Boolean;
```

Purpose

Write absolute disk sectors. *Use this routine with extreme caution!*

Comments

This routine is the counterpart of ReadDiskSectors, and will almost always be used in conjunction with it. When used together, they allow you to read the contents of individual disk sectors into a buffer, alter the contents of the buffer, then write the changes out to disk.

Drive is a number between 0 (= A:) to 25 (= Z:). To convert a drive letter to a drive number, use the following formula:

```
DriveNumber := Ord(UpCase(DriveLetter))-Ord('A');
```

FirstSect is the sector to write to (the first sector on a disk is 0). This value should normally be in the range 0..65535, but if DOS 3.31 or higher is in use, the number can be larger. If DOS 3.3 or lower is in use, the high word of FirstSect is ignored.

NumSects is the number of sectors to write. Buf is the buffer containing the data to be written.

If WriteDiskSectors succeeds, it returns True. Otherwise it returns False and loads DosError (a global variable in the DOS unit) with an error code. See the documentation for ReadDiskSectors for a list of the possible error codes.

WriteDiskSectors is not supported in DOS protected mode due to problems in some DPMI server software.

Example

```
var
  Drive : Byte;
  Buffer : array[1..2048] of Byte;
begin
  Drive := Ord(DefaultDrive)-Ord('A');
  if not ReadDiskSectors(Drive, 0, 1, Buffer) then begin
    WriteLn('Error ', DosError, ' reading from ', DefaultDrive);
    DosError := 0; Exit;
  end;
  {--edit contents of Buffer--}
  if not WriteDiskSectors(Drive, 0, 1, Buffer) then begin
    WriteLn('Error ', DosError, ' writing to ', DefaultDrive, '!');
    DosError := 0; Exit;
  end;
end;
```

See Also

GetDiskInfo

ReadDiskSectors

C. TPENHKBD: Enhanced Keyboard Interface

Although "enhanced keyboards" (the ones with the 12 function keys rather than 10) have quickly become standard equipment on new machines, there are still relatively few programs that fully support them. There are three main reasons for this. First, in order to read all the keys on an enhanced keyboard you have to use special BIOS function calls. Second, before you can use these function calls, you have to be sure that an enhanced keyboard is installed, and there's no standard means of doing so. Third, it is difficult to write programs that work equally well on both "old-style" and enhanced keyboards, because many of the special key combinations recognized by the enhanced keyboard, such as <AltEsc> and <CtrlTab>, are not recognized by the old-style keyboard.

TPENHKBD solves these problems by detecting and activating the enhanced keyboard on machines that have one, and simulating an enhanced keyboard on machines that don't. When TPENHKBD is installed, the enhanced scan codes will be returned by all procedures that read from the keyboard: ReadKey (CRT/TPCRT), ReadKeyWord (TPCRT), and any other routine that uses the standard DOS/BIOS keyboard services. There are no procedures that you need to call here: the unit installs and deinstalls itself automatically, and its operation is completely transparent to any program that uses it.

How TPENHKD Works

TPENHKD performs its magic by installing two interrupt handlers. The first takes over interrupt $16, the BIOS keyboard interrupt, and traps all calls to the BIOS requesting keyboard input. If an enhanced keyboard is installed, the interrupt handler will reroute any calls to the standard keyboard functions ($0, $1, and $2) to the corresponding enhanced keyboard functions ($10, $11, and $12).

The second interrupt handler takes over interrupt $9, the hardware interrupt that is generated each time that a key is pressed or released. It checks for certain key combinations that are not normally returned by the BIOS. If one of these combinations is found, it circumvents the BIOS keyboard handler and inserts the appropriate key code (see the table below) into the keyboard buffer.

Key Codes

The following table summarizes the key codes that are returned if a given key or combination of keys is pressed while TPENHKBD is in use. All numbers are in hex and indicate the value that would be returned by a call to the ReadKeyWord routine in TPCRT. The high byte of the word is the scan code, and the low byte is the ASCII character. If you are using the ReadKey function (in CRT or TPCRT) to obtain keyboard input, and the character it returns is #0 (representing the low byte of the key code), a second call to ReadKey would return the scan code (the high byte). Any

keys not listed here are returned in the usual manner. See the Notes following the table for the key to the special symbols that appear next to some of the key codes.

	Plain	Shift	Control	Alt	
F11	8500%	8700%	8900%	8B00%	
F12	8600%	8800%	8A00%	8C00%	
Esc	011B	011B	011B	0100*	
Backquote	2960	297E	2960	2900*	
Backspace	0E08	0E08	0E7F	0E00*	
Tab	0F09	0F00	9400*	A500*	
Left Bracket	1A5B	1A7B	1A1B	1A00*	
Right Bracket	1B5D	1B7D	1B1D	1B00*	
Backslash	2B5C	2B7C	2B1C	2B00*	
Semicolon	273B	273A	—	2700*	
Quote	2827	2822	—	2800*	
Enter	1C0D	1C0D	1C0A	1C00*	
Comma	332C	333C	—	3300*	
Period	342E	343E	—	3400*	
Slash	352F	353F	—	3500*	
Insert	5200	5230	9200*	A200#	
Del	5300	532E	9300*	A300#	
Home	4700	4737	7700	9700#	
End	4F00	4F31	7500	9F00#	
PgUp	4900	4939	8400	9900#	
PgDn	5100	5133	7600	A100#	
Up	4800	4838	8D00*	9800#	
Down	5000	5032	9100*	A000#	
Left	4B00	4B34	7300	9B00#	
Right	4D00	4D36	7400	9D00#	
Pad-Asterisk	372A	!!!!	!!!!	3700#	(numeric keypad keys)
Pad-Minus	4A2D	4A2D	8E00*	4A00#	
Pad-Plus	4E2B	4E2B	9000*	4E00#	
Pad-5	4C00*	4C35	8F00*	9C00#	

Notes:

— These keystrokes are ignored.
* These keystrokes are not normally returned by the non-enhanced keyboard.
\# These keystrokes are not normally returned by the enhanced keyboard.
!!!! These keystrokes control printscreen and print echoing. They cannot simulate the effect of the enhanced keyboard, which has the Asterisk and PrtSc keys separated.
% The F11 and F12 keys are returned only if they actually exist on the keyboard.

With a few exceptions, the TPENHKBD unit returns results identical to those of the enhanced keyboard BIOS. One difference makes the enhanced keyboard simpler to use in a program: those scan words that normally contain $E0 in the low byte to indicate that the key is specific to the enhanced keyboard (like the dedicated cursor keys) will have the low byte cleared to 0 by TPENHKBD.

Keys marked with # are normally ignored by the enhanced BIOS. By defining these numeric keypad keys when <Alt> is pressed, the usual ability to enter arbitrary keys on the numeric keypad is disabled. Within applications, the ability to use Alt-shifted

arrow keys will generally be more useful. Note, however, that you can still enter arbitrary keys on the numeric keypad by pressing both <Alt> and <LeftShift> simultaneously while entering the number. For example, to enter #196 you would hold down the <Alt> and <LeftShift> keys with your left hand, enter 196 on the numeric keypad with your right hand, then release <Alt> and <LeftShift>.

Note that PREF, our memory resident quick reference chart, displays all the special key codes returned by TPENHKBD. If you pop up PREF and look at the column labeled 'Extended', you'll see that these special keys are displayed in high intensity or white, making it easy to distinguish standard and non-standard key codes.

Miscellaneous Issues

The test used by TPENHKBD to detect the presence of an enhanced keyboard is not foolproof. It is guaranteed to work on PS/2's, and as far as we know it will never report that an enhanced keyboard is installed when there isn't one, but it does sometimes fail to detect that an enhanced keyboard is installed when there is. To deal with these situations, we provide a tiny program called EKFIX.COM that you may distribute with your own programs if you wish. This utility simply toggles a single bit of a byte in low memory, a bit that is supposed to be set if an enhanced keyboard is installed (bit 4 of the byte at $40:$96). The message displayed by the program indicates whether the bit flag was turned on (enabled) or off (disabled).

TPENHKBD should generally appear early in a program's USES list, preferably first. This is particularly important for TSRs. If you are using TPENHKBD in a program that also uses TPTSR, TPENHKBD must come before TPTSR in the USES list to insure that interrupt vectors can be restored in the proper order.

Important warning: Because this unit takes over interrupt $9, it can crash the system if SideKick is installed and a TSR that takes over interrupt $9 has previously been loaded after SideKick. (This is why Borland has always recommended that SideKick be loaded last.) You must exercise similar caution in using this unit in programs that must take over interrupt $9 for other reasons. This is not a concern for programs that use TPTSR, since it automatically disables the code in SideKick that causes problems for interrupt $9 handlers, nor is it usually a concern with SideKickPlus.

It is a concern in programs using TPENHKBD that execute other programs (by calling Exec, ExecDos, etc.). In such a program, use the following strategy:

```
{restore keyboard vectors grabbed by TPENHKBD}
RestoreKbdVectors;
{--- execute other program ---}

{reenable TPENHKBD's interrupt handlers}
InitKbdVectors;
```

Following these steps will insure that no problems will occur if SideKick is loaded ahead of your program and the program that you are executing also takes over interrupt $9.

TPENHKBD: Declarations

Constants

```
EnableEnhanced : Boolean = True;
```

Setting this variable to False temporarily disables TPENHKBD's interrupt handlers.

Variables

```
FiltersEnhanced : Boolean;
```

Set to True by TPENHKBD if the BIOS exhibits filtering behavior. On some machines, this apparent "feature" of the enhanced keyboard BIOS causes int $16 to filter enhanced keys that it finds in the keyboard buffer when the non-enhanced calls are made. Fixing it requires fooling the BIOS by making the enhanced BIOS calls even though no enhanced keyboard is actually installed. Our int $16 handler takes account of the filtering when necessary by using the enhanced calls. Note that the test used by TPENHKBD to detect the filtering behavior has one side effect: it clears the keyboard buffer when the program is initialized.

```
HasEnhancedKbd : Boolean;
```

This variable is set to True when an enhanced keyboard has been detected. This variable is interfaced primarily for informational purposes. Altering its value may lead to undesirable results.

InitKbdVectors / RestoreKbdVectors

Declaration

```
procedure InitKbdVectors;
```

Purpose

Save and setup interrupt vectors taken by TPENHKBD.

Comments

This routine is intended to be used only in conjunction with RestoreKbdVectors. It is called automatically by TPENHKBD's initialization code, and it must not be called again unless RestoreKbdVectors is called first. Calling InitKbdVectors without calling RestoreKbdVectors first causes a system crash the next time that a key is pressed.

InitKbdVectors is interfaced for one purpose only: to allow you to temporarily restore the interrupt vectors grabbed by TPENHKBD (by calling RestoreKbdVectors) and later reactivate its interrupt handlers. See the section on "Miscellaneous Issues," above, for details.

Declaration

```
procedure RestoreKbdVectors;
```

Purpose

Restores original vectors for int's $09 and $16.

Comments

This routine restores the interrupt vectors that are grabbed by TPENHKBD when a program begins. These vectors are restored automatically when the program ends, but in some cases it may be necessary to restore them manually by calling this procedure. The most common case is a TSR that has just called DisableTSR to remove itself from memory.

Once RestoreKbdVectors has been called, there is no way to reactivate the unit's interrupt handlers. If you just need to disable the interrupt handlers temporarily, set the global variable EnableEnhanced to False, then back to True when you want to reenable the interrupt handlers.

Example

```
uses TpEnhKbd, TpTsr;
...
if DisableTSR then
  RestoreKbdVectors
else
  {display error message--not safe to unload the TSR} ;
```

Here RestoreKbdVectors is called as soon as it is certain that the TSR will be removed from memory. Notice that TPENHKBD comes *before* TPTSR in the USES statement.

See Also

DisableTSR (TPTSR)

6. Interrupts and TSRs

A. Overview

Surely two of the most daunting tasks faced by PC programmers are the writing of safe Interrupt Service Routines (ISR's for short) and--perhaps the ultimate challenge--rock solid programs that Terminate and Stay Resident (hence the term TSR) without causing trouble. Although Turbo Professional can't guarantee your resident programs to be bug-free or insure that your ISR's don't use too much stack space, it does provide a host of routines to help get you where you need to go, fast.

Along with TPCRT, one of the most fundamental units in the Turbo Professional library is TPINT. Its primary purpose is to manage your ISR's for you. When you call DOS to take over an interrupt vector, it asks you for the number of the interrupt and the address of your ISR. The InitVector function in TPINT does too, but it also asks you to specify a *handle* (much like a DOS file handle). This handle is what allows it to keep track of things for you, and once you've called InitVector you can usually forget about your ISR. If you do need to restore an interrupt vector yourself, you can just call RestoreVector with the handle, and TPINT will supply DOS with the correct interrupt number and the address of the old service routine. But if your program ends normally, or even if it aborts with a runtime error, you don't need to worry about the interrupt vectors you've captured, because TPINT will automatically restore all of them for you.

Turbo Pascal 4.0 introduced a great convenience in the form of the "interrupt procedure," which saves and restores all the CPU registers for you, initializes the DS (Data Segment) register, and sets up a stack frame so that you can treat these registers as parameters. This in itself is sufficient to allow you to write simple ISR's without resorting to assembly language, but in many cases it isn't nearly enough. So the secondary purpose of the TPINT unit is to provide assistance in the actual writing of your ISR's, especially large ones that need lots of stack space. It contains routines to allocate and deallocate alternate stacks, to switch stacks from inside your ISR, to quickly return from the interrupt no matter how many procedures you've called, to "chain" to the previous interrupt handler, and more.

If writing and debugging interrupt service routines is like climbing jagged rocks, developing a professional-quality TSR can be likened to scaling a glacier. The light is blinding, the surface slippery. But as the wealth of memory resident demonstration programs included in the package demonstrates, Turbo Professional can help you make quick work of this mountainous task. You've probably never felt the need to write a pop-up "Hello World!" program, but if you did, how would you do it? How

many lines of source code would it take? This is how you could do it using Turbo Professional:

```
{$S-,I-,R-}
{$M 2048,0,655360}
program HelloWorld;
uses
  Dos, TpInt, TpTsr;
const
  HotKey = $080F;              {code for Alt-Tab}
  WaitForDos = True;           {don't pop up in the middle of DOS}

  {$F+}
  procedure PopupEntryPoint(var Regs : Registers);
  begin
    WriteLn('Hello world!');
  end;
  {$F-}

begin
  {define the popup}
  if DefinePop(HotKey, @PopupEntryPoint, Ptr(SSeg,SPtr), WaitForDos) then begin
    WriteLn('Pop up loaded, press <Alt><Tab> to activate.');
    {Make popup routines active.}
    PopupsOn;
    {try to go resident}
    if not TerminateAndStayResident(ParagraphsToKeep, 0) then {};
  end;

  {if we get here we failed}
  WriteLn('Unable to go resident');
end.
```

You won't want to pass this program around to your friends, of course, but what can you expect for a mere 30 lines of source code?

As you can see, writing TSRs with Turbo Professional is remarkably easy, and what you get for your efforts are solid programs that won't pop up in the middle of DOS if you don't want them to; that won't destroy the interrupted program's stack; that can pop up on virtually any combination of keystrokes you might want; that won't load themselves twice (if you're willing to make a couple of simple procedure calls before going resident); that won't abort due to DOS critical errors; and that can easily be unloaded from memory if it's safe to do so. (All of the Turbo Professional demonstration programs that terminate and stay resident offer this option, by the way.) There's even a facility that allows your TSRs to communicate with each other and an error handler that tries to prevent your most innocent runtime errors from violently crashing the system.

The TP8087 unit is intended primarily to help you write resident programs that use the numeric coprocessor (an 8087, 80287, or 80387 chip) for real number

calculations. The Save8087 procedure allows you to save the coprocessor's registers when your TSR pops up, and Restore8087 to restore them on exit. TP8087 also offers two routines to control how the 80x87 handles errors. The Exceptions8087 procedure lets you disable exception interrupts or reenable them. The Error8087 function returns the current coprocessor status, allowing you to determine whether a calculation error occurred in the preceding sequence of instructions. TP8087 also supports the Turbo Pascal floating point emulator.

One problem that every PC programmer faces sooner or later is the need to trap DOS critical errors, the kind that produce the infamous "Abort, Retry, Ignore" messages. Although Turbo Pascal traps these errors for you, the method they use can make it difficult to determine what error actually occurred. The TPINT24 unit provides a critical error handler that can help you distinquish clearly between critical and non-critical errors. Note that the popup dispatcher in the TPTSR automatically installs a DOS critical error handler identical to the one in TPINT24 before calling popups that use DOS services.

B. TPINT: ISR Management

As indicated at the start of this chapter, the TPINT unit serves three important functions, the most important of which is ISR (Interrupt Service Routine) management.

By that we mean simply that TPINT keeps track of your interrupt handlers for you. The normal procedure for capturing an interrupt vector in Turbo Pascal is to 1) call the GetIntVec routine in the DOS unit, 2) save the current vector, 3) call SetIntVec to point the vector to your ISR, and 4) restore the old vector when your program ends. Step 4 is the hard one, because a program can end for any number of reasons and at any number of points. In many cases, TPINT lets you reduce the four steps to one:

```
const MyIsrHandle = 16;
...
if InitVector(IntNumber, MyIsrHandle, @MyIsr) then {};
```

TPINT will save the current vector for you, change it to point to MyIsr, and restore the old vector when your program ends.

TPINT can keep track of up to 20 (MaxISRs) ISR's, far more than any program is likely to have, and each one must be assigned a unique handle. A handle is simply a number used to identify a particular ISR, and in most programs you can use any number between 1 and 20. Be advised, however, that two other units in Turbo Professional that use TPINT reserve particular handles for their own use. TPTSR greedily demands numbers 1-11 and 13, and TMACRO uses #12. Since the number of ISR handles reserved for other units may grow in the future, we recommend using only handle numbers 16 and up, especially in your TSRs.

The other principle function of TPINT is to provide routines that simplify the process of writing interrupt service routines in Pascal. All of these routines depend on two things: 1) Turbo Pascal's "interrupt procedures" and 2) variables of a data type defined in TPINT, IntRegisters. If you've studied the Turbo Pascal manual's discussion of interrupt procedures, you know that it allows you to treat the CPU registers as parameters by declaring the interrupt procedure thus:

```
procedure MyIsr(Flags, CS, IP, AX, BX, CX, DX, SI, DI, DS, ES, BP : Word);
  interrupt;
begin
end;
```

In order to use the procedures we're talking about, you would simply change this declaration to:

```
procedure MyIsr(BP : Word); interrupt;
var
  Regs : IntRegisters absolute BP;
begin
end;
```

This buys you two things. First, you can now refer to the "half registers" (AH, AL, BH, BL, etc.) using "Regs.AH", "Regs.AL", and so on. Second, you can now treat the various registers as a group, and pass all of them together as a single parameter to a procedure.

Now, assuming that you've set your ISR up as we've described, you can use the five routines that we've been preparing to talk about. IntReturn lets you return from an interrupt from a procedure other than your ISR. For example, you could call a routine like this one:

```
procedure MyIsrHelper(var Regs : IntRegisters);
begin
  {--- do something ---}
  IntReturn(Regs);
end;
```

ChainInt lets you end your interrupt and pass control to the old interrupt vector like so:

```
ChainInt(Regs, ISR_Array[MyIsrHandle].OrigAddr);
```

SwapStackAndCallNear and SwapStackAndCall let you switch to an alternate stack and call another procedure from inside your ISR. When the procedure that you have called returns control, the old stack is restored.

And EmulateInt lets you call the old interrupt vector from inside your ISR without exiting. For example:

```
EmulateInt(Regs, ISR_Array[MyIsrHandle].OrigAddr);
```

Finally, TPINT provides two functions that make it a bit easier to allocate and deallocate alternate stacks. AllocateStack allocates a stack (just an array, really) of a specified size on the heap, and returns a pointer to the end of the array, since the machine stack grows downward. Its companion procedure, DeallocateStack, disposes of a stack allocated in this fashion.

Reminder: As indicated in the Turbo Pascal manual (Inside Turbo Pascal section, under Writing Interrupt Procedures), Turbo Pascal's I/O routines and memory management routines are not reentrant, and cannot be used in an interrupt service routine. DOS isn't reentrant, so DOS services can't be used. And, finally, many Turbo Professional routines are not reentrant. If you need to be certain, check the

source code for the routine: any routine that modifies a global variable declared in a Turbo Professional unit is probably not reentrant.

Constants

```
MaxISRs = 20;
```

The maximum number of interrupt service routines that can be managed by TPINT.

Types

```
Dummy5 = array[1..5] of Word;
```

Used for padding in the declaration for IntRegisters.

```
IntRegisters =
  record
    case Byte of
      1 : (BP, ES, DS, DI, SI, DX, CX, BX, AX, IP, CS, Flags : Word);
      2 : (Dummy : Dummy5; DL, DH, CL, CH, BL, BH, AL, AH : Byte);
  end;
```

A record whose structure mirrors the contents of the stack frame in an interrupt procedure.

```
ISR_Record =
  record
    IntNum : Byte;
    OrigAddr : Pointer;
    NewAddr : Pointer;
    Captured : Boolean;
  end;
```

A record containing information about a particular interrupt service routine. IntNum is the vector that was taken over. OrigAddr is the address of the previous ISR. NewAddr is the address of the current ISR. And Captured is used internally to determine whether or not the vector has already been restored.

Variables

```
ISR_Array : array[1..MaxISRs] of ISR_Record;
```

The global array of records containing information about the ISR's being managed by TPINT.

Declaration

```
function AllocateStack(var P : Pointer; SizeInBytes : Word) : Boolean;
```

Purpose

Allocates a stack of size SizeInBytes.

Comments

This function provides a convenient means of allocating alternate stacks for interrupt service routines and popups. SizeInBytes is the size of the stack to be allocated, and cannot be greater than 65,515. If a stack of the desired size can be allocated, AllocateStack returns a pointer to the end of it in P. If not, the function returns False.

Very important: Do not use FreeMem to deallocate a stack allocated with AllocateStack. Use DeallocateStack instead.

Example

```
if not AllocateStack(MyStack, MyStackSize) then begin
  WriteLn('Insufficient memory. Aborting...');
  Halt(1);
end;
```

Here AllocateStack is called to allocate a stack of MyStackSize bytes. If there is insufficient memory available on the heap, the function returns False and the program aborts.

See Also

DeallocateStack

ChainInt

Declaration

```
procedure ChainInt(var Regs : IntRegisters; JumpAddr : Pointer);
```

Purpose

Restores stack, registers from Regs and 'jumps' to JumpAddr.

Comments

This inline macro allows you to restore the CPU registers pushed onto the stack at the beginning of an interrupt procedure, then pass on the interrupt to the ISR that serviced the vector before you grabbed it. Unlike the JumpToOldIsr macro in TPINLINE.INC, ChainInt is intended primarily for ISR's written in Pascal, rather than inline code, and it can be called from a procedure other than the actual interrupt procedure. (Assuming, that is, that Regs was passed as a VAR parameter to the second procedure.)

Examples

```
if not InitVector($09, MyIsrHandle, @MyIsr) then
  {error};
...
procedure MyIsr(BP : Word); interrupt;
var
  Regs : IntRegisters absolute BP;
begin
  {-- something --}
  ChainInt(Regs, ISR_Array[MyIsrHandle].OrigAddr^);
end;
```

Incorrect! The '^' in 'ISR_Array[MyIsrHandle].OrigAddr^' is not needed.

```
ChainInt(Regs, ISR_Array[MyIsrHandle].OrigAddr);
```

Correct. ChainInt will restore the registers and "jump" to the previous ISR.

See Also

EmulateInt

JumpToOldIsr (TPINLINE)

Declaration

```
procedure DeallocateStack(P : Pointer);
```

Purpose

Frees an allocated stack.

Comments

This routine allows you to easily dispose of a stack allocated with AllocateStack. Note that the size of the stack is not required as a parameter, since AllocateStack will have already stored that information (just above the top of the stack) for DeallocateStack's use.

Very important: Do not use DeallocateStack to dispose of an alternate stack that was not allocated by AllocateStack. Use Dispose or FreeMem instead.

Example

```
if not AllocateStack(MyStackPtr, MyStackSize) then
  {error};
...
DeallocateStack(MyStackPtr);
```

Here DeallocateStack is used to dispose of a stack allocated with AllocateStack.

See Also

AllocateStack

EmulateInt

Declaration

```
procedure EmulateInt(var Regs : IntRegisters; IntAddr : Pointer);
```

Purpose

Emulates an interrupt.

Comments

This routine simulates an actual interrupt by filling the CPU registers with the values in Regs, clearing interrupts, pushing the flags, and calling FAR to IntAddr. Like the CallOldIsr in TPINLINE.INC, it thus allows you to call the ISR that serviced a particular interrupt vector before you grabbed it. Unlike CallOldIsr, however, EmulateInt is intended primarily for ISR's written in Pascal, rather than inline code, and it can be called from a procedure other than the actual interrupt procedure. (Assuming, that is, that Regs was passed as a VAR parameter to the second procedure.)

A second common use of EmulateInt is to call a routine via a *command entry pointer*. See the documentation for the TPTSR unit for details.

Examples

```
if not InitVector($09, MyIsrHandle, @MyIsr) then
  {error};
...
procedure MyIsr(BP : Word); interrupt;
var
  Regs : IntRegisters absolute BP;
begin
  {-- something --}
  EmulateInt(Regs, ISR_Array[MyIsrHandle].OrigAddr^);
end;
```

Incorrect! The '^' in 'ISR_Array[MyIsrHandle].OrigAddr^' is not needed.

```
EmulateInt(Regs, ISR_Array[MyIsrHandle].OrigAddr);
```

Correct. EmulateInt will reload the CPU registers with the contents of Regs, call the previous ISR, then load Regs with the changed contents of the CPU registers.

See Also

CallOldIsr (TPINLINE) ChainInt

Declaration

```
function InitVector(IntNumber, Handle : Byte; UserRoutine : Pointer) : Boolean;
```

Purpose

Sets up an interrupt service routine.

Comments

The central routine in the TPINT unit, InitVector allows you to change an interrupt vector (IntNumber) to point to an interrupt service routine that you have written (pointed to by UserRoutine). The Handle is the number that you want to use to identify the ISR in question. (See the overview at the start of this section for a discussion of handles.) Once you've called InitVector, all you need to worry about is servicing the interrupt. When your program ends, the the old vector will be restored for you automatically, even if a runtime error occurred. If you need to restore the vector before the program ends, you can do so with RestoreVector (or RestoreAllVectors). InitVector returns False only if the Handle specified is greater than MaxISRs or it is already in use.

Examples

```
if not InitVector($09, MyIsrHandle, @MyIsr) then
  {handle already in use, most likely};
```

Here InitVector is used to change interrupt vector $09 to point to MyIsr.

```
RestoreVector(MyIsrHandle);
Code := ExecDos('', True, nil);
if not InitVector($09, MyIsrHandle, @MyIsr) then
  {shouldn't occur};
```

In this example, RestoreVector is called to restore the old interrupt $09 handler before dropping into a DOS shell, and InitVector is called afterward to grab back the vector.

See Also

RestoreAllVectors

RestoreVector

InterruptsOff / InterruptsOn

Declaration

```
procedure InterruptsOff;
```

Purpose

Turn interrupts off.

Comments

This tiny inline macro simply turns interrupts off.

See Also

InterruptsOn

Declaration

```
procedure InterruptsOn;
```

Purpose

Turn interrupts on.

Comments

This tiny inline macro simply turns interrupts on.

Example

```
procedure MyIsr(BP : Word); interrupt;
var
  Regs : IntRegisters absolute BP;
begin
  InterruptsOff;
  {-- something --}
  InterruptsOn;
end;
```

Here interrupts are turned off at the beginning of the ISR, and turned on again at the end.

See Also

InterruptsOff

Declaration

```
procedure IntReturn(var IntRegs : IntRegisters);
```

Purpose

Return from interrupt.

Comments

This routine lets you return from an interrupt from almost anywhere in your program, provided that the current procedure has been passed a VAR parameter of type IntRegisters by the actual ISR. IntReturn is needed only when the stack has been changed, presumably using SwapStackAndCall or SwapStackAndCallNear, or when you need to return from an interrupt while within a procedure called by the actual ISR.

Example

```
procedure NonIsr(var Regs : IntRegisters);
begin
  {-- something --}
  if NeedToReturnFast then
    IntReturn(Regs);
  {-- something --}
end;

procedure MyIsr(BP : Word); interrupt;
var
  Regs : IntRegisters absolute BP;
begin
  if Regs.AX = MagicValue then
    NonIsr(Regs)
  else begin
    {-- something --}
  end;
end;
```

In this example, MyIsr calls NonIsr, passing Regs as a parameter, and NonIsr uses IntReturn to return from the interrupt.

See Also

SwapStackAndCall

SwapStackAndCallNear

RestoreAllVectors / RestoreVector

Declaration

```
procedure RestoreAllVectors;
```

Purpose

Restores all captured interrupt vectors.

Comments

This routine allows you to restore all interrupt vectors that you have captured. It is intended primarily for internal use, since RestoreAllVectors is called for you automatically when your program ends.

Example

```
if not InitVector($09, MyIsrHandle, @MyIsr) then {}
if not InitVector($16, MyOtherIsrHandle, @MyOtherIsr) then {}
...
RestoreAllVectors;
```

See Also

RestoreVector

Declaration

```
procedure RestoreVector(Handle : Byte);
```

Purpose

Restores an interrupt vector to its original value.

Comments

This routine allows you to restore a vector previously grabbed using a call to InitVector. Since any interrupt vectors that you have grabbed are restored for you automatically when your program ends, the primary use for RestoreVector is to restore a vector temporarily. For example, PCALC.PAS calls InitVector each time the popup is activated to temporarily grab interrupt $0 so it can trap arithmetic errors, then calls RestoreVector on exit from the popup to restore the old vector.

Example

```
if not InitVector($09, MyIsrHandle, @MyIsr) then {handle already in use?};
...
RestoreVector(MyIsrHandle);
Code := ExecDos('', True, nil);
if not InitVector($09, MyIsrHandle, @MyIsr) then {};
```

Restore the old interrupt $09 handler before dropping into a DOS shell.

See Also

InitVector

RestoreAllVectors

Declaration

```
procedure SendEOI;
```

Purpose

Send an End Of Interrupt command to the Programmable Interrupt Controller.

Comments

This inline macro sends an EOI (End Of Interrupt) signal to the PIC (Programmable Interrupt Controller). It should be used only in interrupt handlers that are servicing hardware interrupts such as $8 (clock tick) or $9 (keyboard).

Examples

```
procedure Int9(BP : Word); interrupt;
var
  Regs : IntRegisters absolute BP;
  B : Byte;
begin
  {get the scan code for the key press/release}
  B := Port[$60];
  ...
  if not NeedToEatKeyStroke then
    ChainInt(Regs, OldInt9)
  else begin
    {reset the keyboard}
    B := Port[$61];
    Port[$61] := B or $80;
    Port[$61] := B;
    {signal end of interrupt}
    SendEOI;
  end;
end;
```

A fairly realistic application of SendEOI--an interrupt $9 handler that needs to "eat" keystrokes under certain conditions.

SwapStackAndCall

Declaration

```
procedure SwapStackAndCall(Routine, SP : Pointer; var Regs : IntRegisters);
```

Purpose

Switches to stack designated by SP and calls Routine with Regs as a parameter.

Comments

This routine allows you to switch to an alternate stack (pointed to by SP) and call a procedure, passing Regs as a parameter. The Routine to be called should be of the form

```
{$F+}
procedure MyProc(var Regs : IntRegisters);
begin
  {-- something --}
end;
{$F-}
```

Very important: MyProc must be a FAR call from the current ISR. If it is not, you should use SwapStackAndCallNear instead.

Examples

```
{$F+}
procedure NonIsr(var Regs : IntRegisters);
begin
  {-- something --}
  if NeedToReturnFast then
    IntReturn(Regs);
  {-- something --}
end;
{$F-}

procedure MyIsr(BP : Word); interrupt;
var
  Regs : IntRegisters absolute BP;
begin
  if Regs.AX = MagicValue then
    SwapStackAndCall(@NonIsr, MyStackPtr^, Regs)
  else begin
    {-- something --}
  end;
end;
```

Incorrect! The '^' in 'MyStackPtr^' is not needed.

```
SwapStackAndCall(@NonIsr, MyStackPtr, Regs)
```

Correct. SwapStackAndCall will switch to the stack pointed to by MyStackPtr and call NonIsr, passing Regs as a parameter. Notice that NonIsr has been compiled under the FAR model.

See Also

SwapStackAndCallNear

Declaration

```
procedure SwapStackAndCallNear(Routine : Word; SP : Pointer;
                               var Regs : IntRegisters);
```

Purpose

Switches to stack designated by SP and calls Routine with Regs as a parameter.

Comments

Like SwapStackAndCall, this routine allows you to switch to an alternate stack (pointed to by SP) and call a procedure, passing Regs as a parameter. The Routine to be called should be of the form

```
{$F-}
procedure MyProc(var Regs : IntRegisters);
begin
  {-- something --}
end;
```

Very important: MyProc *must* be a NEAR call from the current ISR. If it is not, you should use SwapStackAndCall instead.

Example

```
procedure NonIsr(var Regs : IntRegisters);
begin
  {-- something --}
  if NeedToReturnFast then
    IntReturn(Regs);
  {-- something --}
end;

procedure MyIsr(BP : Word); interrupt;
var
  Regs : IntRegisters absolute BP;
begin
  if Regs.AX = MagicValue then
    SwapStackAndCallNear(Ofs(NonIsr), MyStackPtr^, Regs)
  else begin
    {-- something --}
  end;
end;
```

Incorrect! The '^' in 'MyStackPtr^' is not needed.

```
SwapStackAndCallNear(Ofs(NonIsr), MyStackPtr, Regs)
```

Correct. SwapStackAndCall will switch to the stack pointed to by MyStackPtr and call NonIsr, passing Regs as a parameter. Notice that NonIsr has *not* been compiled under the FAR model.

See Also

SwapStackAndCall

C. TPTSR: TSR Management

The main purpose of the TPTSR unit is to provide a state-of-the-art TSR management system that is easy to use and solid as a rock. By following a few simple guidelines, you should be able to write reliable memory resident programs just as easily as non-resident ones. But before we go into the details of how to do it, let's define a few terms.

A TSR (Terminate and Stay Resident) is a program that, shortly after being loaded, returns control to the operating system (DOS) while remaining in memory. Every TSR contains one or more *popup* routines, which are to be called by the *popup dispatcher* when a particular event occurs. The job of the popup dispatcher may be divided into roughly six parts: 1) monitor the system to watch for events of interest; 2) when such an event occurs, determine which popup needs to be called; 3) based on what is known about the popup, determine whether or not it can be invoked safely; 4) before calling the popup, save the state of the system and take all possible precautions to insure the safety of the popup; 5) call the popup; 6) restore the state of the system. (Note: The order of events described here and in the following paragraph has been intentionally distorted in order to make this complex process easier to understand.)

Now let's look at a simple example, the "Hello World!" program shown at the start of the chapter, so we can see these principles in action. The popup dispatcher monitors the keyboard, waiting for the user to press the <Tab> key at a time when the <Alt> key has already been pressed. When this event occurs, the dispatcher checks its tables and sees that the PopupEntryPoint procedure needs to be called, and that it needs access to DOS services (WaitForDos is True). The dispatcher then checks to be sure that these services can be accessed safely and, assuming that they can, proceeds to create a stable environment for the popup to run in. Finally, PopupEntryPoint is called. Once it has finished, it returns control to the dispatcher, which then restores the state of the system, returns the CPU to the task it was working on when <Alt><Tab> was pressed, and goes back to monitoring the system.

On to the details, which we'll divide into topics.

Number and kinds of popups

By default, TPTSR allows you to define as many as 8 (MaxPopups) popup routines in any one program, each of which falls into one of two categories. The kind you will probably use most frequently is a popup associated with a *hotkey*--more on it in a minute. The other kind of popup is not invoked as a result of any particular event. Instead, you can invoke it whenever you want by calling the SetPopTicker procedure with the handle assigned to the popup (see below) and a clock tick count (TimeOut). This tells the popup dispatcher to start watching for a good opportunity

to call the popup. If TimeOut clock ticks go by without such an opportunity, the popup will not be called. For a good example of such a popup, see PTIME.PAS, which uses one to display an on-screen clock, to sound alarms, and to play back SMACS macros at designated times.

Hotkeys

The more conventional popup is one that is invoked when a specified combination of keys is pressed simultaneously, a combination referred to collectively as a hotkey. In Turbo Professional's terms, a hotkey is actually a single key that is pressed while a particular combination of shift keys is already being pressed. Besides the regular shift keys, <LeftShift> and <RightShift>, the <Alt> and <Ctrl> keys are also treated as shift keys for this purpose. To specify the hotkey for your popup, you need to construct a word value that is bit-mapped as follows:

Bit #	Value	Meaning
15-12	$0000	unused -- must be zero
11	$0800	Alt
10	$0400	Ctrl
9	$0200	Left shift
8	$0100	Right shift
7-0	$00xx	Scan code for main key

For example, let's say that you want your popup to activate when <Alt>, <LeftShift>, and <Z> are pressed. The scan code for <Z> is $2C--that gives us the value for the low byte, bits 0-7. If we add to that the values for <Alt> and <LeftShift> we get $0800 + $0200 + $002D = $0A2D. (PCALC is quite handy for doing hexadecimal arithmetic like this, by the way.)

More on hotkeys

Two other points to keep in mind concerning hotkeys. First, you can define a popup that is activated by a single keystroke. If the high byte of the hotkey code is 0, simply pressing the key whose code is specified in the low byte activates the popup, provided that none of the four shift keys is being pressed at the same time. For example, to pop up on <SysReq>, specify a hotkey code of $0054. You can also use one of the shift keys as the main key. For example, to pop up on <Ctrl><LeftShift> you could specify either $042A or $021D, depending on which of the two keys was to be pressed first. Second, you are limited to one hotkey per popup. Although DefinePop will let you define the same popup twice using different hotkeys, doing so is potentially quite dangerous, and we advise against it.

Popup handles

When you call DefinePopProc to define a popup that is invoked as a result of a call to SetPopTicker, it appears to ask for a "Handle". Unlike the TPSCREEN and TPINT units, which allow you to assign your own handles, TPTSR assigns them for

you. So all you need to do is declare a global byte variable--say, "MyPopupHandle"--and pass it as the parameter. If any handles are still available, DefinePopProc will assign MyPopupHandle a value and return True. When you are ready to invoke your popup, you would then say something like:

```
SetPopTicker(MyPopupHandle, 18);
```

which means "during the next 18 clock ticks, look for a good opportunity to call the popup associated with MyPopupHandle." Popups activated by hotkeys are also assigned handles, but you don't need to know what they are since they are for housekeeping purposes only.

Popup routines

All popups written with Turbo Professional, whether they are activated by the press of a key or by a call to SetPopTicker, should be of the form

```
{$F+}
procedure PopupShell(Regs : Registers);
begin
  {-- body of the popup --}
end;
{$F-}
```

Both the {$F+} directive and the declaration of the Registers parameter are critical. If either is missing, your popup will be called successfully, but you'll crash on exit. When calling DefinePop/DefinePopProc, the "Routine" parameter should be a pointer to the popup, like so: "@PopupShell".

Two points worth noting here. First, the Regs parameter *must* be of type Registers (defined in the DOS unit), *not* of type IntRegisters (defined in TPINT), if you want to be able to *refer to* the values in the registers. Second, if you want to be able to *change* the values in the registers, you should make it a VAR parameter. Changing the values is not recommended, but it is technically possible. Making Regs a VAR parameter will also conserve stack space.

Stacks

Each popup must also have its own stack to work with. This means that the StackPtr parameter passed to DefinePop/DefinePopProc must be different for each popup in the program. Most TSRs consist of a single popup, and in these cases you can almost always use "Ptr(SSeg, SPtr)" as the parameter. But programs that contain multiple popups (such as SMACS.PAS and PTIME.PAS) must allocate space for alternate stacks, either by declaring an array as a global variable or by calling the AllocateStack routine in TPINT, which allocates an array on the heap. In either case, the StackPtr parameter must point to the last element of the array rather than the first. (Note that AllocateStack always returns a pointer to the end of the array.)

Waiting for DOS

Most popups need access to DOS services, but some do not. If you are uncertain, you should play it safe and assume that it does. But if you know for a fact that it doesn't (e.g., if it does something simple like draw text on the screen using the FastWrite routine in TPCRT), you should pass "False" to DefinePop as the DosWait parameter. This will allow you to pop up in many situations where you otherwise could not.

"DOS services," by the way, covers a broad range of territory. All disk I/O, most notably, requires DOS's assistance, but so do lots of other things that might surprise you. The GetDate and GetTime routines in the DOS unit, for example, use DOS calls. Again, when in doubt, assume that DOS is needed.

For further details, see the section on "Playing it safe," below.

Before going resident

One more thing needs to be done before going resident, especially if you are using the TPCRT or CRT units in your program, and you are using Turbo Pascal 4.0. If you look at the source code for any of the memory resident demonstration programs, you'll invariably find the following statements near the end:

```
{$IFDEF Ver40}
SetIntVec($1B, SaveInt1B);
{$ENDIF}
```

What this does is restore the vector for interrupt $1B (generated by the BIOS when <CtrlBreak> is pressed), a vector captured by both CRT and TPCRT. This vector *must* be restored before you go resident. If you are using Turbo Pascal 5.0 or higher, this will be done for you automatically, but it won't if you're using version 4.0. Similarly, if the non-resident portion of your program captures any other interrupt vectors not needed by the resident portion, these vectors must also be restored before going resident.

Going resident

OK, you've defined your popups, now what do you do? In most cases, you would simply call the TerminateAndStayResident function like so:

```
if not TerminateAndStayResident(ParagraphsToKeep, 0) then {error};
```

ParagraphsToKeep determines how much memory your program is currently using, so the above statement would instruct TerminateAndStayResident to release everything you aren't already using to DOS. Usually this is the right thing to do, but if your popups need to allocate space on the heap later on, there won't be any memory left to allocate. So, assuming that you wanted to set aside 4000 bytes (250

paragraphs) of heap space for your popups, you'd do this instead:

```
if not TerminateAndStayResident(ParagraphsToKeep+250, 0) then {error};
```

For an example of a TSR that allocates heap space after going resident, see SMACS.PAS.

Important note: It is not absolutely necessary to set an especially low value for a resident program's maximum heap size (using the {$M} compiler directive). On the contrary, if you need to allocate things on the heap after going resident, it is a good idea to use a very high number, and TerminateAndStayResident move the heap downward for you. The ParasToKeep parameter to TerminateAndStayResident must not request more heap space than is available.

Running your TSR

In case you didn't know it already, TSRs *must* be loaded from the DOS command line. So, if you are accustomed to running your programs from inside the Turbo compiler, you'll have to make some changes in your programming habits. Similarly, most debuggers *cannot* be used to debug TSRs, including our T*Debug*PLUS 4.0, Borland's Turbo Debugger, and the debugger built into Turbo Pascal 5.0 and higher. The debugger that we use for TSRs is Periscope, and we highly recommend it.

I/O errors and DOS critical errors

One of the things that the dispatcher does before it calls a popup that needs access to DOS services is to install a *critical error handler* (like the one in TPINT24) to trap the kind of errors that cause DOS to display the familiar "Abort, Retry, Ignore" message. These are bad news in any program, but they can mean disaster for a TSR. The error handler installed by the dispatcher simply tells DOS to ignore the error (under DOS 2.x) or to fail the call (under DOS 3.0 or higher). This will get you back into your program, but under DOS 2.x the error condition will still exist, so you have to be prepared to deal with it. How? At the top of your program, add the {$I-} directive, so the runtime library won't try to abort when an I/O error occurs. After each I/O operation that could conceivably cause an I/O error, add a call to Int24Result. If it returns anything other than 0, either an I/O error or a DOS critical error occurred, and you should handle it just as you would if you had called IoResult instead.

Runtime errors

Runtime errors can be just as deadly, of course. TPTSR contains an error handler that attempts to salvage the system when runtime errors occur, but this isn't always possible. The good news is that it is *usually* possible if your popup uses an alternate stack (in the data segment or on the heap), so you would be well advised to use one while developing your program. (A finished TSR should not generate runtime

errors. If it does, it isn't finished.) Note that this error handler does not display any error messages, and if it succeeds in its salvage operation, Turbo's error handler, which does display error messages, will never be reached. So if you need to see what error occurred you'll need to write your own error handler by following the steps outlined in the Turbo Pascal manual.

Using TPTSR with TPWINDOW and/or TPMENU
When using either TPWINDOW or TPMENU in programs written with TPTSR, keep in mind that both TPWINDOW and TPMENU make liberal use of the heap. They do most of their memory allocation when windows and menus are first defined, but they also allocate/deallocate some smaller blocks of memory as menus and windows are displayed on/removed from the screen.

In general, this means that you should 1) make all of your calls to MakeWindow, SubMenu, and MenuItem *before* going resident, and 2) set aside at least a few hundred extra bytes of heap space at the time you go resident. For example,

```
if not TerminateAndStayResident(ParagraphsToKeep+25, 0) then {error};
```

This would insure that at least 400 (25*16) bytes will be available on the heap after going resident. See the earlier section on "Going Resident" for a discussion of issues related to the allocation of heap space while resident.

Of course, the same basic guidelines described here apply to the use of *any* unit or routine that needs to allocate heap space, such as TPSORT.

Using TPTSR with TPMACRO
If you use both TPTSR and TPMACRO in your program, TPTSR *must* appear before TPMACRO in the USES list. This is because TPINT restores captured interrupt vectors in reverse order based on their handle numbers (12, 11, ..., 1), *not* based on the order in which the vectors were first captured. Normally this doesn't matter, because most programs capture a particular vector only once. But in cases where the same vector is captured twice--TPMACRO and TPTSR both grab interrupt $16--TPINT imposes an undocumented requirement that the vectors be captured in handle number order (TPTSR uses handle 7 for interrupt $16, TPMACRO uses handle 12).

Stack and heap usage
One difficult task that TPTSR can't do for you is determine how much stack or heap space a particular popup requires. Our best advice is this. During the development stages, allocate more stack and heap space than you think you could possibly need, and don't worry about memory usage. When your program is mostly finished, use the OPSTACK unit found in the Turbo Professional BONUS directory to start

getting accurate measurements of heap and stack usage. Although OPSTACK is designed primarily for use in non-resident programs, it can be adapted for use in TSRs if necessary. See the documentation for OPSTACK (also in the BONUS directory) for instructions.

Some special issues in TSR programming

There are also a few other tasks that TPTSR cannot handle for you. If you are writing a TSR that uses hardware (8087) reals, you must be sure to save and restore the state of the math coprocessor in your popup. The TP8087 unit provides the necessary routines: Save8087 and Restore8087. The TSR manager leaves this job to you because it doesn't know if it is necessary. It does, however, ensure that the interrupt vectors used by the floating point emulator are set properly on entry to your popup.

A similar need arises in TSRs that need to use EMS memory or overlays while popped up. The TPEMS unit provides the routines that you need to save and restore the state of the Expanded Memory Manager so that you can access EMS memory safely: SaveEmsContext and RestoreEmsContext. For an explanation of how to write TSRs that use Turbo Pascal's overlay system, see Appendix C, "Using Overlays with Turbo Professional."

Modules and standard interfaces

Have you ever accidentally tried to load the same TSR twice? Many TSRs aren't equipped to deal with that possibility, and the results of such an oversight usually aren't very pretty. TPTSR makes it easy for you to deal with this potential problem, however, by creating what we call a *standard interface*. This interface allows any TSR that uses it not only to determine if it's already loaded, but also to communicate with all other TSRs (or *modules*) that use it. (More on that point in a moment.) Briefly, this is what you need to do to avoid the problem: Assign each TSR a unique name, a string of say 5-10 characters. Before going resident, call ModuleInstalled to see if you're already resident:

```
const ModuleName : string[5] = 'MYTSR';
...
if ModuleInstalled(ModuleName) then begin
  WriteLn('Already installed. Aborting...');
  Halt;
end;
```

To make this work, of course, you need to "install" the module before going resident the first time:

```
InstallModule(ModuleName, nil);
```

This lets your program find itself, and other programs find it.

Command entry pointers

Why would other programs want to find you? You may have noticed "nil" in the call to InstallModule above. The second parameter to InstallModule, CmdEntryRoutine, is the address of a procedure that other programs can call to have you execute a particular function for them. Any program can locate this routine by means of this *command entry pointer*, like so:

```
EntryPointer := ModulePtrByName('MYTSR');
if EntryPointer = nil then
  {MYTSR is not loaded}
else
  EntryPointer := EntryPointer^.CmdEntryPtr;
```

Your CmdEntryRoutine should be of the form:

```
procedure MyCmdEntryRoutine(BP : Word); interrupt;
var
  Regs : IntRegisters absolute BP;
begin
  with Regs do begin
    {--handle function calls--}
  end;
end;
```

Why is it an interrupt procedure? Two reasons. First, this allows it to be called easily using the EmulateInt routine in TPINT. Second, it allows you to pass parameters using the CPU registers, just as you do with Intr and MsDos. For example, let's say that MyCmdEntryRoutine does something special if Regs.AH := 1. You'd call it like this:

```
var Regs : IntRegisters;
...
Regs.AH := 1;
EmulateInt(Regs, EntryPointer);
```

The possible uses for this facility are unlimited. For an example of a TSR that has a CmdEntryRoutine, see SMACS.PAS, and for examples of programs that call this routine see PTIME.PAS and PCALC.PAS.

Unloading a TSR

One very nice feature that many TSRs (including lots of well-known commercial ones) lack is a means for the user to unload it. Here again, TPTSR makes it easy for you to offer your users this option. From inside your popup, just call DisableTSR. If it returns True, your TSR will be disabled (unloaded from memory) as soon as you leave your popup. If it returns False, that means that it is very definitely unsafe to unload the TSR, so it won't be done. For a definition of "unsafe," see the documentation for DisableTSR. As mentioned earlier, all of the memory resident demonstration programs offer this option, so you can consult any or all of them for examples.

It is also possible to write a program that can unload itself from the DOS command line. Space does not permit an adequate explanation of how to do this, but SMACS.PAS demonstrates the technique. If you wish to add this capability to your own programs, study the source code for SMACS, starting in the 'Install' procedure, where it checks for the '-U' command line parameter.

Popups everywhere

In the preceding paragraphs, it was assumed that you will be writing popups only for memory resident programs. This doesn't have to be the case, however. You can also use popups in "normal" programs. You could use one to display an on-screen clock (see PTIME.PAS for an example), to implement an "auto-save" feature in a text editor--the possibilities are endless. You *must* keep in mind, however, the problems of reentrancy discussed in the overview for the TPINT unit (Chapter 6, Section B), since the popup will likely be called during the execution of your own program's code.

Getting along with SideKick

SideKick can cause all sorts of problems for other TSRs, but TPTSR helps you out here in two ways.

First, during initialization, TPTSR checks to see if SideKick is in memory. If it is, it tries to patch the resident copy of SideKick in such a way as to allow other TSRs to be loaded safely after SideKick. The only known disadvantage to patching SideKick and then loading your TSR above it is that SideKick will then no longer be able to keep itself from being loaded twice. We recognize that this is a sneaky trick, and if you'd prefer that your memory resident programs be mild mannered participants in the TSR wars, you can simply undefine the conditional compilation directive 'ThwartSideKick', found in TPDEFINE.INC, and recompile TPTSR.

Second, if SideKick's presence is detected, and TPTSR cannot find the proper patch point, it will set a global flag called 'SideKickLoaded' that you can check before going resident if you want to display an error message. Checking this flag is optional, but TPTSR will refuse to let you terminate and stay resident if SideKickLoaded is True.

TPTSR can also detect the presence of SideKickPlus, but it will leave the SideKickLoaded flag set to False if it is found, since it normally does not cause the same kinds of problems for other TSRs that SideKick does. You should almost always be able to safely load a TSR after SideKickPlus.

Playing it safe

Before calling a popup that needs access to DOS services, the popup dispatcher takes several special precautions to insure the safest possible environment for the

popup to run in: (1) DOS's current BREAK level is saved, then "relaxed" to insure that pressing <CtrlBreak> will not kill the popup. (2) Similarly, interrupts $1B and $23 are both pointed to IRET instructions. (3) The current DTA (Disk Transfer Address) and PSP (Program Segment Prefix) are saved, and the ones that were current when the TSR was loaded are restored. (4) As noted earlier, a DOS critical error handler is installed to allow the popup to detect and recover from these errors. When the popup returns control to the dispatcher, the system is restored to the state it was in before these steps were taken.

Technical notes for hackers

TPTSR always monitors the following interrupts: $05 (PrtSc), $08 (Clock tick), $09 (Keyboard), $13 (BIOS disk read/write), $16 (BIOS keyboard), $25 (DOS absolute disk read), $26 (DOS absolute disk write), $28 (DOS multitasking) and $33 (mouse driver). Interrupt $09 is monitored to watch for hotkeys; when a hotkey is pressed, the "pop ticker" is set to 36 (two seconds). The handlers for interrupts $08 and $28 together represent the "popup dispatcher": if the pop ticker is non-zero, they check to see if it is safe to pop up and, if so, invoke the appropriate popup. The other interrupts are merely "filtered," to insure that popups will not be invoked while one of these interrupts is in progress. If the popup is being run in the OS/2 "compatibility box," four other interrupts are also filtered: $10 (BIOS video interrupt), $14 (BIOS communications), and $17 (BIOS printer). That's enough information to give you some idea of how popups are handled. If you want to know more, you'll need to study TPTSR.ASM.

TPTSR: Declarations

Types

```
IfcPtr = ^IfcRecord;
```

Pointer to an IfcRecord.

```
IfcRecord = record
              NamePtr : ^String;
              Version : Word;
              CmdEntryPtr : Pointer;
              PrevIfc,
              NextIfc : IfcPtr;
            end;
```

A record containing data about a particular module. NamePtr points to the name of the module. Version is the version number of the interface. If not nil, CmdEntryPtr points to a routine that can be called by other programs. PrevIfc and NextIfc point to the previous and following entries in the linked list of modules.

Constants

```
Int5Handle = 1;
Int8Handle = 2;
Int9Handle = 3;
Int10Handle = 4;
Int13Handle = 5;
Int14Handle = 6;
Int16Handle = 7;
Int17Handle = 8;
Int25Handle = 9;
Int26Handle = 10;
Int28Handle = 11;
Int33Handle = 13;
```

These are the ISR handles used by TPTSRs interrupt handlers. (Handle 12 is used by TPMACRO.)

```
MaxPopups = 8;
```

Maximum number of popup routines.

```
SideKickLoaded : Boolean = False;
```

Set to True by initialization code if SideKick is already loaded.

Variables

```
PopTickerPtr : ^Word;
```

Points to a variable hidden in TPTSR that indicates if we are trying to pop up. The value of PopTickerPtr^ tells the popup manager how many more times to attempt to activate the popup (attempts are made 18 times a second). This variable is intended primarily for internal use.

```
PopupInUse : array[1..MaxPopups] of Boolean;
```

These flags are used to keep track of which popups are currently active. They are intended primarily for internal use.

```
PopupsEnabledPtr : ^Boolean;
```

Points to a hidden variable that indicates if popups are enabled. This variable is intended primarily for internal use.

```
PopupToCallPtr : ^Byte;
```

Points to a hidden variable that contains the handle of the popup waiting to be called if PopTickerPtr^ is non-zero. This variable is intended primarily for internal use.

```
ThisIfc : IfcRecord;
```

Record containing data about a particular module.

ChangeHotKey

Declaration

```
function ChangeHotKey(OldHotKey, NewHotKey : Word) : Boolean;
```

Purpose

Change a popup's hotkey from OldHotKey to NewHotKey.

Comments

This routine allows you to change the hotkey for a particular popup. The only restriction is that the scan code portion of the new hotkey must not conflict with the hotkey for any other popup. For example, you could change a hotkey from <Alt><Tab> to <Ctrl><Tab>, but you could not change it to <Ctrl><Space> if another popup had <Alt><Space> for a hotkey.

Example

```
const
  OldHotKey = $0829;  {<Alt><~>}
var
  NewHotKey : Word;
begin
  NewHotKey := $040F; {<Ctrl><Tab>}
  if not ChangeHotKey(OldHotKey, NewHotKey) then
    {hotkeys not the same, and new one already in use};
end;
```

Changes the hotkey for the popup associated with <Alt><~> to <Ctrl><Tab>.

Declaration

```
function DefinePop(HotKey : Word; Routine, StackPtr : Pointer;
                   DosWait : Boolean) : Boolean;
```

Purpose

Defines a popup routine associated with a particular hotkey.

Comments

Unlike DefinePopProc, this routine is used to define a popup that can be activated by HotKey. Routine is a pointer to the popup. StackPtr points to the end of the stack to be used by the popup. And DosWait indicates whether or not the popup needs access to DOS services. DefinePop returns False if no more Popup handles are available.

Examples

```
const
  MainHotKey : Word = $0829; {Alt+ '~'}
  ReaderHotKey : Word = $0839; {Alt + ' '}
var
  ReaderStack : array[1..2048] of Byte;
...
if not DefinePop(ReaderHotKey, @ReaderPop, @ReaderStack[1], True) then
  Abort(NoCanPop);
```

Incorrect! '@ReaderStack[1]' is the address of the *start* of the alternate stack, not the end of it.

```
if not DefinePop(ReaderHotKey, @ReaderPop, @ReaderStack[SizeOf(ReaderStack)],
  True) then
    Abort(NoCanPop);
```

Correct.

```
if not DefinePop(MainHotKey, Ptr(CSeg,Ofs(MainPop)), Ptr(SSeg,SPtr), True) then
  Abort(NoCanPop);
```

Probably OK, but the following is better:

```
if not DefinePop(MainHotKey, @MainPop, Ptr(SSeg, SPtr), True) then
  Abort(NoCanPop);
```

See Also

DefinePopProc DeletePop

DefinePopProc

Declaration

```
function DefinePopProc(var Handle : Byte; Routine, StackPtr : Pointer) : Boolean;
```

Purpose

Defines a routine that can be triggered by calling SetPopTicker.

Comments

Unlike DefinePop, this routine is used to define a popup that can be activated by calling SetPopTicker. Routine is a pointer to the popup. StackPtr points to the end of the stack to be used by the popup. It is assumed that the popup needs access to DOS services. Handle is a number assigned to the popup by DefinePopProc; this number is needed in order to call SetPopTicker. DefinePopProc returns False if no more Popup handles are available.

Examples

```
var
  ClockPopHandle : Byte;
  ClockStack : array[1..512] of Byte;
...
if not DefinePopProc(ClockPopHandle, @ShowClock, @ClockStack[1]) then
  Abort(ClockErrorMsg);
```

Incorrect! '@ClockStack[1]' is the address of the *start* of the alternate stack, not the end of it.

```
if not DefinePopProc(ClockPopHandle, @ShowClock,
  @ClockStack[SizeOf(ClockStack)]) then
    Abort(ClockErrorMsg);
```

Correct.

See Also

DefinePop
DeletePopProc
SetPopTicker

Declaration

```
procedure DeletePop(HotKey : Word);
```

Purpose

Deletes the popup routine associated with HotKey.

Comments

This routine lets you deactivate a popup defined with DefinePop. Its handle may then be used for other popups.

Examples

```
const
  MyHotKey = $0439;
...
DeletePop($0829);
DeletePop(MyHotKey);
```

See Also

DefinePop

DeletePopProc

Declaration

```
procedure DeletePopProc(Handle : Byte);
```

Purpose

Deletes a popup routine defined with DefinePopProc.

Comments

This routine lets you deactivate a popup defined with DefinePopProc. Its handle may then be used for other popups.

Example

```
var
  MyPopupHandle : Byte;
...
if not DefinePopProc(MyPopupHandle, @MyPopup, @MyStack[SizeOf(MyStack)]) then
  {error};
{-- something --}
DeletePopProc(MyPopupHandle);
```

See Also

DefinePopProc

DeletePop

DisableTSR

Declaration

```
function DisableTSR : Boolean;
```

Purpose

Disable TSR by restoring interrupt vectors and releasing memory.

Comments

This routine lets you disable (or "unload") a TSR in many cases. Before disabling the program, it checks to see if any other program has grabbed one of the interrupt vectors taken over by the program. If one has, then it's not safe to disable the TSR, and the function will return False. Otherwise, it will restore all the interrupt vectors that have been grabbed and release all of the program's memory to DOS. It does *not* actually halt the program. If DisableTSR returns True, your program should simply exit from the popup normally.

Warning: DisableTSR must not be called from a popup that does not have access to DOS services.

Examples

```
begin
  {-- start of popup --}
  if DisableTSR then
    Halt;
  {-- end of popup --}
end;
```

Incorrect! You should never Halt while inside a popup.

```
begin
  {-- start of popup --}
 if DisableTSR then
    Exit;
  {-- end of popup --}
end;
```

Correct.

See Also

RemovePops

DosBusyFlag / DosCriticalFlag

Declaration

```
function DosBusyFlag : Byte;
```

Purpose

Returns current value of DOS busy flag.

Comments

This routine, which is intended to be called from within a popup, indicates whether or not DOS was active at the time the popup was activated. If the value is 0, DOS was not active; otherwise, it was. Assuming that the popup was designated as needing access to DOS services when it was declared, the value returned should be non-zero only if the popup was activated while the user was at the DOS prompt.

Example

```
if (DosBusyFlag <> 0) then begin
  StuffString(^['TURBO.EXE'^M);
  Exit;
end;
```

Assuming that it was executed from inside a popup, this would stuff an <Esc>, the letters 'TURBO.EXE', and a carriage return into the keyboard buffer, then return to the DOS prompt.

Declaration

```
function DosCriticalFlag : Byte;
```

Purpose

Returns current value of DOS critical flag.

Comments

This routine, which is intended to be called from within a popup, indicates whether or not a DOS critical error was pending at the time the popup was activated. If the value is 0, DOS was not critical; otherwise, it was.

Note that under certain versions of DOS this value could conceivably be equal to the value returned by DosBusyFlag, and not have any special meaning. To see why, examine the InitTsrPtrs routine in TPTSR.ASM.

InitPops / InstallModule

Declaration

```
procedure InitPops;
```

Purpose

Initializes TPTSRs data structures.

Comments

This procedure must be called before any other popup related routines. Since it is called automatically when the program begins, you do not need to call it yourself unless you are reinstalling popups after previously calling RemovePops.

Example

```
RemovePops;
{--something--}
InitPops;
```

See Also

RemovePops

Declaration

```
procedure InstallModule(var ModuleName : string; CmdRoutine : Pointer);
```

Purpose

Installs this program as a resident module that can be located and accessed by other programs.

Comments

ModuleName is a string that uniquely identifies a TSR module. CmdRoutine, if not nil, must point to a pseudo-interrupt service routine that other programs can call using EmulateInt. InstallModule adds the module to the linked list of resident modules that use the standard interface, thus enabling any module to find a duplicate copy of itself, as well as other modules, by calling ModuleInstalled. The module can be detached from the list by calling UninstallModule.

Example

```
const MyTsrName : string[5] = 'MYTSR';
...
InstallModule(MyTsrName, nil);
```

See Also

EmulateInt (TPINT)
ModuleInstalled (TPINT)
UninstallModule (TPINT)

Declaration

```
function Int24Result : Word;
```

Purpose

Returns the latest result from the int $24 handler.

Comments

This routine is functionally identical to the Int24Result in the TPINT24 unit. The only difference is that TPTSRs critical error handler is installed only while a popup is in progress. This error handler cannot be disabled. See the documentation for Int24Result (TPINT24) for examples.

See Also

Int24Result (TPINT24)

Declaration

```
function ModuleInstalled(var ModuleName : string) : Boolean;
```

Purpose

Returns true if ModuleName is installed.

Comments

This function allows you to determine whether a particular module is installed. You can use it to prevent loading your own TSRs twice, as well as to check for the presence of another program (such as SMACS) that can be called on to execute special functions via its command entry pointer.

Examples

```
const MyModuleName : string[5] = 'MYTSR';
      OtherModuleName : string[10] = 'SUPER MACS';
...
if ModuleInstalled(MyModuleName) then begin
  WriteLn('Already loaded. Aborting...'); Halt(1);
end;
if not ModuleInstalled(OtherModuleName) then begin
  WriteLn('SMACS not loaded. Aborting...'); Halt(1);
end;
```

See Also

ModulePtrByName

ModulePtrByName / ParagraphsToKeep

Declaration

```
function ModulePtrByName(var ModuleName : string) : IfcPtr;
```

Purpose

Returns a pointer to the IfcRecord for the module named ModuleName or nil.

Comments

Although superficially similar to ModuleInstalled, the main purpose for this routine is to allow you to locate the command entry pointer for another module.

Example

```
const SmacsName : string[10] = 'SUPER MACS';
var P : IfcPtr;
  EntryPointer : Pointer;
begin
  P := ModulePtrByName(SmacsName);
  if P <> nil then EntryPointer := P^.CmdEntryPtr
  else EntryPointer := nil;
end;
```

This demonstrates how you would get the command entry pointer for SMACS.

See Also

ModuleInstalled

Declaration

```
function ParagraphsToKeep : Word;
```

Purpose

Returns # of paragraphs currently used for code, data, stack, and heap.

Comments

This routine calculates the total amount of memory (in paragraphs) that your program is using. Its only real use is to help you calculate the value to be passed to TerminateAndStayResident (the ParasToKeep parameter).

Examples

```
if not TerminateAndStayResident(ParagraphsToKeep, 0) then {error}
```

Go resident with no extra heap space.

```
if not TerminateAndStayResident(ParagraphsToKeep+1024, 0) then {error}
```

Go resident with 16K (1,024 paragraphs * 16 bytes/paragraph) of heap space available.

See Also

TerminateAndStayResident

Declaration

```
procedure PopupsOff;
```

Purpose

Turns popups off.

Comments

This routine allows you to avoid conflicts among popups in programs that contain more than one. Popups can be reenabled with PopupsOn.

Example

```
{$F+}
procedure MyPopup(var Regs : Registers);
begin
  PopupsOff;
  {--body of popup--}
  PopupsOn;
end;
{$F-}
```

Here other popups are locked out until MyPopup ends.

Declaration

```
procedure PopupsOn;
```

Purpose

Turns popups on.

Comments

This routine enables popups. In TSRs, it should be called just prior to going resident. In conjunction with PopupsOff, it also allows you to avoid conflicts among popups in programs that contain more than one.

Examples

```
PopupsOn;
if not TerminateAndStayResident(ParagraphsToKeep, 0) then
  {error};
```

A standard sequence in a TSR: enable popups, then go resident.

```
{$F+}
procedure MyPopup(var Regs : Registers);
begin
  PopupsOff;
  {--body of popup--}
  PopupsOn;
end;
{$F-}
```

Here other popups are locked out until MyPopup ends.

RemovePops / SetPopTicker

Declaration

```
procedure RemovePops;
```

Purpose

Removes all popups from the system and releases all associated interrupt vectors.

Comments

This routine is intended primarily for internal use. It is called automatically when a program ends. Note that this is not the same function performed by DisableTSR, which effectively halts a TSR.

See Also

DisableTSR

InitPops

Declaration

```
procedure SetPopTicker(Handle : Byte; TimeOut : Word);
```

Purpose

Sets the pop ticker for the indicated procedure, clears interrupts, then returns.

Comments

This tells the popup dispatcher to look for a good opportunity to call the popup associated with Handle. If TimeOut clock ticks go by without such an opportunity, the popup will not be called. (There are roughly 18.2 clock ticks per second.) *Very important:* No range checking is performed by SetPopTicker, so Handle absolutely *must* be a valid handle number (1..MaxPopups) returned by DefinePopProc.

Examples

```
SetPopTicker(1, 36);
```

Incorrect! Don't use a constant as the Handle parameter, even if there's only one popup in your program.

```
SetPopTicker(MyPopupHandle, 36);
```

Correct. This means "try for the next two seconds to activate the popup associated with MyPopupHandle" (returned by DefinePopProc).

See Also

DefinePopProc

TerminateAndStayResident / UninstallModule

Declaration

```
function TerminateAndStayResident(ParasToKeep : Word; ExitCode : Byte) : Boolean;
```

Purpose

Terminate and stay resident, freeing up all our memory except ParasToKeep.

Comments

This routine 1) releases all memory allocated to a program except ParasToKeep (number of paragraphs to keep, usually calculated by ParagraphsToKeep); 2) fixes up the heap so that memory can be allocated dynamically after going resident; 3) restores the interrupt vectors grabbed by the Turbo runtime library; and 4) calls the DOS Keep function ($31), passing it ParasToKeep and ExitCode (almost always 0) as parameters. TerminateAndStayResident returns False if it is unable to release the program's memory.

Examples

```
if not TerminateAndStayResident(ParagraphsToKeep, 0) then {error}
```

Go resident with no extra heap space.

```
if not TerminateAndStayResident(ParagraphsToKeep+1024, 0) then {error}
```

Go resident with 16K (1,024 paragraphs * 16 bytes/paragraph) of heap space available.

See Also

ParagraphsToKeep PopupsOn

Declaration

```
procedure UninstallModule;
```

Purpose

Uninstalls the module from the linked list of modules.

Comments

This routine simply removes the current module from the linked list of modules. It is intended primarily for internal use (by DisableTSR, for example).

See Also

DisableTSR InstallModule

D. TP8087: 8087 Support for TSRs

As noted earlier, TP8087 is intended primarily for TSRs that use 8087 reals (Single, Double, Extended, Comp) for real number calculations. It does contain a function (Installed8087) to test for the presence of a math chip, but it is rarely needed because Turbo Pascal 4.0 generates initialization code to test for the chip's presence when you compile with the {$N+} directive, and Turbo 5.0 or higher provides floating point emulation. Save8087, Restore8087, and Reinit8087, however, are of critical importance if your TSR needs to use the coprocessor. Although the popup dispatcher in TPTSR takes care of the CPU registers for you, it does not save the coprocessor's registers. So, if your popup uses 8087 reals, the main routine should look something like this:

```
{$F+}
procedure PopupEntryPoint;
var
  Save8087Regs : SaveBuffer8087;
begin
  Save8087(Save8087Regs);
  Reinit8087;
  ...
  Restore8087(Save8087Regs);
end;
{$F-}
```

These three simple procedure calls insure that your popup's use of the coprocessor won't cause problems for the program it interrupted.

If your TSR or ISR uses the floating point emulator, you'll need to use two other routines: ReinitEmulator, which reinitializes Turbo Pascal 5's floating point emulator, and UseEmulator, which determines whether or not the floating point emulator is being used. The most important thing to keep in mind when using them is that ReinitEmulator should be called only after you have switched to an alternate stack. (In a popup, the stack switch occurs automatically; in an interrupt service routine, you need to use SwapStackAndCall or SwapStackAndCallNear.) What ReinitEmulator does is to reinitialize the 230-byte data area starting at SSeg:0 that is used by the emulator, something that *must* be done before you can execute any arithmetic instructions using the emulator. Obviously, your alternate stack must be at least 230 bytes larger than would otherwise be necessary, in order to make room for the emulator's data.

What follows is a simple program that demonstrates the use of these routines. The program captures interrupt $1C (called at every clock tick) and performs floating point math each time an interrupt occurs.

```
{$N+,E+,F-}
program UseEmulator;
uses  Tp8087, TpInt;
const D : Double = 2.0;
var
  I : Word;
  UseEmu : Boolean;
  AltStack : array[1..2048] of Byte;

  procedure Dummy(var Regs : IntRegisters);
  var
    SB : SaveBuffer8087;
    D : Double;
  begin
    if UseEmu then
      {reinitialize emulator}
      ReinitEmulator
    else begin
      {save 8087 registers}
      Save8087(SB);
      {reinitialize 8087}
      Reinit8087;
    end;

    {do some math}
    D := 1.0;
    D := D * 2.0;
    if not UseEmu then
      {restore 8087 registers}
      Restore8087(SB);
  end;

  procedure Int1C(BP : Word); interrupt;
  const
    Active : Boolean = False;
  var
    Regs : IntRegisters absolute BP;
  begin
    if not Active then begin
      Active := True;
      SwapStackAndCallNear(Ofs(Dummy), @AltStack[2048], Regs);
      Active := False;
    end;
  end;

begin
  UseEmu := UseEmulator;
  if UseEmu then
    WriteLn('Using emulator...')
  else
    WriteLn('Using 80x87 chip...');
  {install $1C handler}
  if not InitVector($1C, 20, @Int1C) then
    Halt;
  WriteLn('Multiplying...');
  for I := 1 to 2000 do
    if D*D <> 4.0 then begin
```

```
      WriteLn('Error at ', I);
      Halt(1);
    end;
  WriteLn('Done...');
end.
```

A similar strategy is required for a TSR. Before going resident, you would call UseEmulator and store the result in a global variable. Within your popup routine(s), you would then check this variable: if the emulator is in use, you would call ReinitEmulator on entry to the popup; if not, you would call Save8087 and Reinit8087 on entry, and Restore8087 on exit.

By default, Turbo Pascal will produce a fatal runtime error whenever a numeric coprocessor calculation goes awry. Divide by zero, numeric overflow, square root of a negative argument--these and other errors will stop your program in its tracks. Even the TPERRHAN unit cannot help because by the time its error handler gains control, the 80x87 internal stack has been corrupted irreversibly.

Fortunately, TP8087 offers the means to solve this problem with just two simple routines. The Exceptions8087 procedure takes one boolean parameter which programs the coprocessor to produce exceptions (which lead to runtime errors) or to report errors via the coprocessor status word and numeric results. The second routine, Error8087, lets your program poll the coprocessor status word to see whether the preceding stream of calculations has produced a numeric error. The Error8087 routine returns a result similar to that of the standard IoResult function: any non-zero value implies an error (which can be decoded for meaning), and the process of reading Error8087 resets the status register for further calculations.

It is important to note here that TP8087 behaves slightly differently depending on how it is compiled. If it is compiled with Turbo Pascal 4.0, or if it is compiled with version 5.0 or higher and the $N- compiler directive is set, all the routines in the unit use 8087 instructions. If it is compiled with version 5, and the $N+ compiler directive is set, all the routines use the floating point emulator in the SYSTEM unit. So the Save8087 routine, for example, would save either the coprocessor's registers or the emulator's registers, depending on whether a math chip were installed or not. In the remainder of the chapter, all references to the 8087 should be assumed to refer also to Turbo Pascal's floating point emulator where appropriate.

Types

```
SaveBuffer8087 =
  record
    Control  : Word;     {control word}
    Status   : Word;     {status word}
    Tag      : Word;     {tags for stack}
    InstPtr  : LongInt; {instruction pointer}
    OpPtr    : LongInt; {data pointer}
    Regs8087 : array[0..7] of Extended; {stack}
  end;
```

Used to store data needed to save and later restore the state of the math coprocessor using Save8087 and Restore8087.

Error8087

Declaration

```
function Error8087 : Word;
```

Purpose

Return (and clear) the error status of the 80x87.

Comments

This function can be used to check for errors after arithmetic operations performed with the math coprocessor. It is useful only if you have programmed the coprocessor not to produce exception interrupts using the Exceptions8087 procedure. It returns the contents of the six lowest bits in the coprocessor's status word, those which indicate error conditions, and resets these flags to enable further processing.

If the function result is 0, no error occurred. Otherwise, you can determine what exception was generated by testing individual bits in the word that is returned:

Bit	Val	Meaning
5	32	Precision lost (result is not exact–had to be rounded)
4	16	Underflow error
3	8	Overflow error
2	4	Divide by zero error
1	2	Denormalized operand (result is so small the significand has leading zeros)
0	1	Invalid operation (e.g., SQRT or LN of negative number)

Note: Bit 5 will be set quite regularly in situations that you probably wouldn't consider an error, as would bit 1. If you aren't interested in these two errors, you can simply mask out these bits by ANDing the function result with $1D.

Example

```
Exceptions8087(False);
...
Extended3 := Extended1 * Extended2;
ErrorCheck := Error8087 {and $1D} ;
```

The call to Error8087 loads ErrorCheck with the contents of the coprocessor's status word and clears its error flags.

See Also

Exceptions8087

Declaration

```
procedure Exceptions8087(On : Boolean);
```

Purpose

Turn exception interrupts on or off.

Comments

This procedure allows you to program a math coprocessor to produce exception interrupts, which generate runtime errors, or to report arithmetic errors using the coprocessor's status word. Since the 80x87 produces exception interrupts by default, the routine thus gives you a means of recovering from otherwise fatal runtime errors. Once you have disabled exception interrupts, you must check for errors manually by polling Error8087.

Note: The FINIT and FNINIT instructions used by routines such as Installed8087 when testing for the presence of an 80x87 will reprogram the coprocessor to generate exception interrupts. If you execute a DOS shell (using ExecDos or Turbo's Exec routines) after disabling exceptions, you should be sure to call Exceptions8087 after the Exec call, in case the program that you executed reprogrammed the coprocessor.

Examples

```
Exceptions8087(False);
...
Extended3 := Extended1 * Extended2;
ErrorCheck := Error8087;
```

The call to Exceptions8087 disables exception interrupts, making it necessary to call Error8087 after the arithmetic operation.

```
Exceptions8087(False);
...
Code := ExecDos('', True, nil);
Exceptions8087(False);
```

The second call to Exceptions8087 insures that the coprocessor is programmed properly after the Exec call.

See Also

Error8087

ExecDos (TPDOS)

Installed8087 / Reinit8087

Declaration

```
function Installed8087 : Boolean;
```

Purpose

Returns True if an 80x87 coprocessor is installed.

Comments

This function allows you to determine whether a particular machine has a math coprocessor installed. Since the Turbo runtime library has its own test for the presence of an 80x87, and refuses to load programs that need it if none is installed, Installed8087 is rarely needed. It is most useful, perhaps, in installation programs used to configure programs that require a coprocessor. As noted elsewhere, this function has the side effect of resetting the coprocessor to its default state, thereby enabling exception interrupts. See the entry for the Exceptions8087 procedure.

Example

```
if not Installed8087 then begin
  WriteLn('The program being installed requires a math coprocessor!');
  Halt(1);
end;
```

Such code might be found in an installation program used to install a program that requires a math coprocessor.

See Also

Exceptions8087

Declaration

```
procedure Reinit8087;
```

Purpose

Reinitialize the 8087 chip.

Comments

This routine reinitializes the 80x87 chip. It should be called just after a call to Save8087.

Example

See the introduction to this section for examples.

See Also

Save8087

Declaration

```
procedure ReinitEmulator;
```

Purpose

Reinitialize the emulator.

Comments

This routine reinitializes the floating point emulator. It should be called only if UseEmulator returns True.

Example

See the introduction to this section for an example.

See Also

UseEmulator

Declaration

```
procedure Restore8087(var SaveBuf : SaveBuffer8087);
```

Purpose

Restores the 80x87 registers from the save buffer.

Comments

This procedure is useful primarily in popup routines written with TPTSR. It allows you to restore the state of the coprocessor (i.e., its registers and stack) so that arithmetic operations performed by the popup do not affect the program that was interrupted. Each call to Restore8087 must be preceded by a call to Save8087, which saves the state of the coprocessor.

Example

```
procedure MyPopup;
var
  SaveBuf : SaveBuffer8087;
begin
  Save8087(SaveBuf);
  {-- rest of popup routine --}
  Restore8087(SaveBuf);
end;
```

Here Restore8087 is called on exit from a popup routine to restore the state of the coprocessor.

See Also

Save8087

Save8087 / UseEmulator

Declaration

```
procedure Save8087(var SaveBuf : SaveBuffer8087);
```

Purpose

Saves the 80x87 registers in the save buffer.

Comments

This procedure is useful primarily in popup routines written with TPTSR. It allows you to save the state of the coprocessor so that the popup can use it without affecting the program that was interrupted. Each call to Save8087 must be followed later with a call to Restore8087, which restores the state of the coprocessor.

Example

```
{$F+}
procedure MyPopup;
var
  SaveBuf : SaveBuffer8087;
begin
  Save8087(SaveBuf);
  {-- rest of popup routine --}
  Restore8087(SaveBuf);
end;
{$F-}
```

Here Save8087 is called on entry to a popup routine to save the state of the coprocessor.

See Also

Restore8087

Declaration

```
function UseEmulator : Boolean;
```

Purpose

Return True if ReinitEmulator should be used rather than Reinit8087.

Comments

This routine returns True if the floating point emulator is present *and* emulation is enabled. The most common use for UseEmulator is to determine if ReinitEmulator should be used rather than Reinit8087.

Example

See the introduction to this section for an example.

See Also

ReinitEmulator

E. TPINT24: Critical Error Handler

TPINT24 is considered to be an obsolete unit and should not be used by new programs. The reason is that Turbo Pascal now installs its own critical error handler, which returns critical error codes through the standard IoResult function.

TPINT24 also adds a DOS critical error handler to your programs, one that automatically installs itself when your program starts and uninstalls itself when it ends. To use TPINT24 in your programs, you need to take just three steps. First, add TPINT24 to the USES clause at the start of your program. Second, turn I/O checking off using the {$I-} compiler directive. Third, after each I/O operation that might produce an error, add a call to the Int24Result function. That is, every place that you would normally call IoResult, you would now call Int24Result instead. What it returns is a byte-coded word, with the DOS critical error code in the high byte, and IoResult in the low byte. The other two routines in the unit, Int24On and Int24Off, allow you to temporarily uninstall the error handler.

Technical note: Under DOS 2.x, TPINT24's error handler will tell DOS to IGNORE the critical error, because this is the only response that will allow your program to recover gracefully from a critical error. Under DOS 3.0 and higher, it tells DOS to FAIL the call that generated the error, to insure that the error is caught.

When compiled in protected mode, TPINT24 simply uses Borland Pascal's int $24 handler rather than installing its own. The Int24Result function calls IoResult and maps the error codes it returns to the values normally returned by Int24Result.

Int24Off / Int24On

Declaration

```
procedure Int24Off(RestoreDosHandler : Boolean);
```

Purpose

Restore DOS's critical error handler or Turbo's.

Comments

This procedure uninstalls TPINT24's critical error handler. If RestoreDosHandler is True, DOS's critical error handler is restored in its place. If it is False, the critical error handler in the Turbo runtime library is restored instead.

Example

See SMACS.PAS for an example of a case where Int24Off/Int24On are needed. You do not need to call Int24Off when your program ends, since that is done for you automatically.

See Also

Int24On

Declaration

```
procedure Int24On;
```

Purpose

Install DOS critical error handler.

Comments

This procedure reinstalls TPINT24's critical error handler. It does not need to be called at the start of your program, because that is done for you automatically.

See Also

Int24Off

Declaration

```
function Int24Result : Word;
```

Purpose

Returns DOS critical error code in high byte, IoResult in low byte.

Comments

This function returns IoResult in the low byte of its result, and a DOS critical error code (see list below) in the high byte. In order for this function to work, I/O checking must be turned off with the {$I-} compiler directive, and Int24Result must be checked religiously after each I/O operation that might generate an error.

Special note: Although it is sometimes possible to catch printer errors (off line, out of paper, etc.) on some machines, detection of these errors is not guaranteed, and should not be counted on.

Critical error codes:

$01 Invalid drive number
$02 Drive not ready
$03 Unknown command
$04 CRC (Cyclical Redundancy Check) error
$05 Bad request structure length
$06 Seek error
$07 Disk format not recognized
$08 Sector not found
$09 Printer out of paper
$0A Write error
$0B Read error
$0C General failure
$0D Attempt to write to write-protected disk (mapped to $0D from $00)

Example

```
Assign(F, FileName);
Reset(F);
if Int24Result <> 0 then begin
  WriteLn('Error opening ', FileName);
  Exit;
end;
```

The call to Int24Result checks to see if either an I/O error or a DOS critical error occurred while trying to open F.

7. Sorting

A. Overview

Sorting is a common task performed by many programs. Uses for sorting range from listing addresses alphabetically to computing statistics on sets of numbers.

Turbo Professional provides a sort routine that is fast and easy to use. With it, you can sort up to 65,535 data elements of any type. All elements must fit into normal memory, however, since paging to disk is not supported. The TPSORT unit is quite compact, using less than 2000 bytes of code space. It implements a non-recursive quicksort that also minimizes use of stack space.

If you have seen Borland's Database Toolbox sort system, using the TPSORT unit will be a snap since the two are organized in a similar fashion. TPSORT differs primarily in implementation. As mentioned above, TPSORT does not support paging to disk--instead it provides a faster, more compact sort of adequate capacity for almost every situation. TPSORT has a special mode for sorting strings--instead of allocating space for the maximum length of a string, it allocates only the actual length. By using procedure pointers, TPSORT also makes the sort completely dynamic, that is, one set of sort routines can sort many data types within the same program.

TPSORT exports three routines: Sort, PutElement, and GetElement. Your program will call Sort first. Sort requires several parameters, as described in detail in the following sections. Two of the parameters tell Sort about the elements to be sorted--their size and the maximum number to expect. Three parameters provide addresses of routines which Sort will call during the course of execution--one routine which will load elements into Sort's data structures, another which Sort will call to compare pairs of elements, and a final one to return the sorted results to your program.

The routines you provide to load and retrieve elements will call the other two functions exported by TPSORT. For each element to be sorted, your load routine will call PutElement to enter the data. For each sorted element retrieved, your routine will call GetElement.

The following example demonstrates the process of sorting a set of integers:

```
program SortInt;
  {-Sort a random set of integers}
uses
  TpSort;
const
  NumToSort = 1000;
var
  Status : SortStatus;

{$F+}
procedure GetInts;
  {-Load the sort data structures with elements to be sorted}
var
  N : word;
  Value : integer;
begin
  for N := 1 to NumToSort do begin
    Value := Random(MaxInt);
    if not PutElement(Value) then
      {Sort error occurred - most likely out of memory}
      exit;
  end;
end;

function Less(var X, Y) : Boolean;
  {-Compare two integers}
var
  XI : integer absolute X;
  YI : integer absolute Y;
begin
  Less := (XI < YI);
end;

procedure WriteInts;
  {-Write the sorted output}
var
  Value : integer;
begin
  while GetElement(Value) do
    WriteLn(Value);
end;
{$F-}

begin
  Status := Sort(NumToSort, SizeOf(Integer), @GetInts, @Less, @WriteInts);
  case Status of
    SortSuccess : WriteLn(ElementsSorted, ' elements sorted');
    SortOutOfMemory : WriteLn('Insufficient memory');
    SortTooManyElements : WriteLn('Too many elements to sort');
  end;
end.
```

Refer to the sections following for further details.

For another example of how to use the Turbo Professional sort system, see the demonstration program BSORT.PAS.

TPSORT and Memory Management
Normally TPSORT uses a special kind of Mark/Release technique to allow rapid deallocation of heap space without destroying the Turbo memory manager's free list. This technique works fine so long as the caller of the TPSORT routines does not need to perform heap allocation or deallocation while the sort is in progress--that is, within the user-defined procedures of the sort. If you do need to do so, you will have to modify the source code for TPSORT. We have made this change a simple one, however: all you need to do is un-define the conditional compilation directive FastDispose in TPDEFINE.INC, then recompile TPSORT.

The FastDispose option does not work reliably with Turbo Pascal version 6 and version 7, nor is it feasible to make it work. Therefore FastDispose is automatically disabled under Turbo Pascal versions 6 and 7, regardless of the setting in TPDEFINE.INC.

Common Mistakes
Roughly 90% of the technical support calls we have received concerning TPSORT have been the result of one or both of two common mistakes: 1) the user-routine routines called by Sort have not been declared FAR with {$F+}, or 2) they are local to another procedure. In order for Sort to work properly, the user-routine routines *must* be global (non-nested) procedures compiled under the FAR model. Please verify that this is the case if you experience any difficulty with TPSORT.

B. TPSORT: In-memory Sort Routine

This section describes all of the types, variables, constants, procedures and functions found in the interface section of the TPSORT unit.

Types

```
SortStatus =
  (SortSuccess,                {Successful sort}
   SortOutOfMemory,            {Insufficient memory}
   SortTooManyElements         {Attempt to sort more elements than requested}
  );
```

The Sort function returns a result of the type SortStatus. The calling application should check this result to determine the outcome of the sort.

Variables

```
ElementsSorted : Word;
```

When a sort has been completed, the ElementsSorted variable will have been initialized to contain the actual number of elements that were sorted.

Declaration

```
function GetElement(var X) : Boolean;
```

Purpose

Get next element from the sorted array.

Comments

Call GetElement to retrieve each element in sorted order after a sort has been completed. Your application will have the opportunity to call GetElement within the retrieval routine whose address you pass to the sort function.

Because TPSORT does not know the type of your data, GetElement's VAR parameter is untyped. It is your responsibility to provide a variable of the proper type. TPSORT knows how many bytes to pass back based on a parameter passed to the Sort function.

GetElement returns True as long as there are further elements to retrieve. Once it returns False, the parameter X is undefined.

Example

```
while GetElement(s) do
  WriteLn(s);
```

This example simply writes sorted elements (of any simple type) to the screen.

See the BSORT.PAS program for further examples of using GetElement.

See Also

Sort

PutElement

Declaration

```
function PutElement(var X) : Boolean;
```

Purpose

Put a sort element into the sort array.

Comments

Call PutElement to add each new element to TPSORT's internal sort structure. Your application will have the opportunity to call PutElement within the loader routine whose address you pass to the Sort function.

Because TPSORT does not know the type of your data, the parameter to PutElement is an untyped VAR parameter. As such, you cannot pass a constant or expression to PutElement. However, the parameter can be any type of variable such as a simple numeric element, a pointer to another structure, or a variable length string. TPSORT knows how many bytes to transfer based on the initial call to Sort.

PutElement returns True when an element is successfully added to the sort array. If it returns False, your application should stop adding elements and exit the loader routine. PutElement will return False if insufficient heap space is available for the new element, or if the number of elements exceeds the maximum allotted during the call to Sort.

Examples

```
while not Eof(f) do begin
  ReadLn(f, s);
  if not PutElement(s) then
    exit;
end;
```

The example reads lines from a text file and adds each one to the sort array. If PutElement returns False, the loader routine is exited.

```
for i := 1 to NumPointers do
  if not PutElement(P[i]) then
    exit;
```

In this example, P is an array of pointers to another data structure, perhaps a record. The pointers themselves are added to the sort rather than the full records (to save memory space). The application's element comparison function will dereference the pointers to perform the comparison. See the BSORT.PAS program for an example of this technique.

See Also

Sort

Declaration

```
function Sort(Elements : Word; ElementSize : Word;
              GetElements : Pointer; LessFunc : Pointer;
              PutElements : Pointer) : SortStatus;
```

Purpose

Sort a group of elements.

Comments

Sort is the main routine of TPSORT. An application will call Sort prior to calling any of the other routines.

Sort is designed to be completely configurable at runtime. Within memory limitations, it can sort up to 65,535 elements of any type, calling user-defined functions for element comparison, input and output. Sort keeps all elements in memory. This choice provides faster operation than a virtual sort and adequate capacity for most tasks.

The first two parameters to Sort tell it how many elements to expect, and the size of each element. The Elements parameter specifies the maximum number to expect. It is no problem if fewer elements are actually available. If more elements are encountered, a sort error occurs.

In all but one special case, ElementSize specifies the number of bytes in each sort element. This size is best obtained by using Turbo's built-in operator SizeOf. For example, SizeOf(pointer) returns 4, and SizeOf(MyDataType) returns the number of bytes in a given data type. Sort provides a special case for strings. If you specify an ElementSize of 0, Sort interprets each element as a Turbo string and allocates memory for just the size required by each string.

Programs using TPSORT must be structured in a certain way as exemplified by the simple example at the beginning of this chapter. The application must define three routines whose addresses are passed to Sort as pointers.

```
{$F+}
procedure GetSortElements;
{-Load elements from user data into the Sort routine}
begin
  ...
  if not PutElement(x) then
    exit;
  ...
end;

procedure PutSortElements;
{-Return sorted elements to the user program}
begin
  ...
  while GetElement(x) do
    {use the element x} ;
  ...
end;

function Less(var X, Y) : boolean;
begin
  {Return boolean for X < Y}
end;
{$F-}
```

Note that the three routines must be of the same type (function or procedure) and take the same parameters as shown (the names of the routines are unimportant), and they *must* be global procedures compiled under the FAR model. The address of each routine is passed to Sort in the three pointer parameters. For this example, Sort would be called as follows:

```
Status := Sort(Elements, ElementSize, @GetSortElements, @Less,
               @PutSortElements);
```

The '@' operator tells the compiler to pass the address of the named routine to the Sort function.

Sort will call your GetSortElements routine after performing some error checking and initialization tasks. GetSortElements in turn calls PutElement to put each element into TPSORT's internal data structure. When GetSortElements returns, Sort actually performs the sorting operation. Whenever Sort needs to compare two elements to place them into sorted order, it calls your Less function. The Less function compares the elements in any desired way depending on the data type, and returns True if the first element X is "less than" the second element Y. After sorting is complete, Sort calls your PutSortElements routine which calls GetElement to get each element back in sorted order. When PutSortElements returns, Sort deallocates the memory it has used and returns a sort status to the original calling program. See the SortStatus type for the codes Sort uses.

Examples

See the example program at the beginning of this chapter, and the BSORT.PAS program for a more extensive example.

8. Memory Allocation

A. Overview

Much has been written about "the 640K barrier" imposed by MS-DOS, and most of us have encountered it ourselves. Sophisticated software needs plenty of memory for code and especially data, but what can you do when you run out of room? Fortunately, there are two alternatives to using normal RAM.

One of these, *extended memory*, is available only on '286 and '386 machines. Extended memory exists above the one megabyte mark and is accessible only by switching these machines into protected mode.

The other alternative, *expanded memory*, is available on any IBM-compatible machine equipped with a board designed in accordance with the Expanded Memory Specification, or EMS for short. EMS memory is made accessible by the Expanded Memory Manager (EMM) software driver, and several companies have developed drivers that allow extended memory to emulate expanded memory. Unfortunately, Turbo Pascal, like most compilers, does not give provide you any way to use either extended or expanded memory to store data. Turbo Professional does.

Note that the extended and expanded memory units of Turbo Professional are not intended for DOS protected mode applications written with Borland Pascal 7.0. You can use the standard New and GetMem commands to allocate extended memory in this case.

The TPEMS unit contains a variety of routines for working with expanded memory. It allows you to determine whether or not EMS memory is installed and, if so, how much is currently available. You can allocate and deallocate memory, and you can also save and restore the state ("context") of the expanded memory manager, in case you need to access EMS memory from inside a TSR, for example.

We should probably point out here that TPEMS is designed to work with version 3.x or higher of the Expanded Memory Specification. It does not support any of the newer functions added in EMS 4.0.

EMS memory is allocated in 16K blocks known as *pages*. This makes it quite useful for storing medium-sized arrays, linked lists, and binary trees, but it is less than ideal for general purpose storage of smaller pieces of data that need to be accessed quickly on a regular basis. If your dynamic storage needs are minimal, you would be better off allocating memory in normal RAM using New or GetMem. Although

EMS memory is also less than ideal for storing large arrays, Turbo Professional offers a solution to this problem in the related unit, TPEARRAY. Using the routines in this unit, you can easily work with arrays of up to 32 megabytes. See the documentation for TPEARRAY in Chapter 10, Section C for more details.

Similarly, the TPEXTMEM unit gives you all the tools you need to use extended memory for your data storage needs. These routines allow you to access extended memory under DOS without colliding with VDISK.SYS, MicroSoft's RAMDRIVE.SYS, or with any other program compatible with either of the two. Note, however, that TPEXTMEM cannot be used when an XMS memory manager such as HIMEM.SYS, 386MAX, or QEMM386 is installed. Use the BONUS unit OPXMS in this case.

To access extended memory, you need to take the following steps:

1. Find out how much extended memory is available by calling ExtMemAvail.
2. If the amount of memory available is sufficient, use GetExtMem to allocate the total amount that you will use.
3. Once the memory has been allocated, use MoveExtMem to move data from normal RAM to extended memory and back again.

As you may have guessed already, TPEXTMEM contains extended memory counterparts for many of Turbo Pascal's built-in routines: MemAvail, GetMem, FreeMem, Mark, Release, and Move. Although the names are similar, there are important differences between them, and a few simple guidelines must be kept in mind when using these routines.

First and most important is the fact that extended memory must be allocated and deallocated in sequential order. If you allocate three blocks of memory, and release the middle block by calling FreeExtMem, all memory above the first block will appear to be available. (The reasons for this restriction will be immediately obvious if you study the source code for TPEXTMEM.)

The second most important guideline to remember is that all addresses in extended memory are referenced using *linear*, rather than *segmented*, notation. For example, the segmented address $0040:$0012 corresponds to the linear address $00000412. The slight disadvantage of this method is that, when moving data between normal and extended memory, you must translate segmented addresses into linear addresses, or *huge pointers*, by calling the PtrToHuge function. The great advantage of it is that huge pointers are nothing more than long integers, so you can create arrays much larger than 64K in extended memory and use long integers for indexing into them. For example, if HP is a huge pointer to a 0-based array, then you can calculate the address of the *n*'th element (HPn) thus:

```
HPn := HugePtr( LongInt(HP) + (N * SizeOf(Element)) );
```

Third, GetExtMem allows you to allocate memory only in 1K blocks (1,024 bytes). As with EMS memory, this makes extended memory less convenient to use for general purpose storage, but it also means that you can allocate several megabytes of contiguous memory with a single call to GetExtMem. In addition, the memory manager in older versions of RAMDRIVE.SYS can handle only 95 blocks of extended memory, at least 1 of which will always be used as a RAM disk. Although TPEXTMEM allows you to allocate more than 94 blocks (so long as RAMDRIVE.SYS is not present), you should not count on having access to more than that if you want your programs to be compatible with RAMDRIVE.

Fourth, the MoveExtMem procedure that is used to transfer data between normal and extended memory does not allow you to move data in bytes, only in words (two bytes). This means that you must be extra careful in choosing your data structures, and that you will sometimes need to pad record types, for example, with dummy bytes to insure that their sizes are even numbers. The one benefit of this restriction is that you can move up to 128K of contiguous data with a single call to MoveExtMem.

Fifth, in order to accommodate RAMDRIVE.SYS, most of the routines in TPEXTMEM require a little over 1K of stack space for local variables. You should keep this in mind when using the unit in programs with restricted stack space.

A final important consideration to keep in mind is that TPEXTMEM works only with DOS versions 3.0 or higher. The fact that TPEXTMEM cannot be used to allocate memory when running under DOS 2.x is irrelevant, since 80286 and 80386 machines (the only kind that might have extended memory installed) cannot run DOS 2.x.

For a good example of how to use the routines in TPEMS and TPEXTMEM in your programs, see the demonstration program DIFF.PAS. DIFF is a utility program that compares the contents of two text files and generates a list of the differences between them. When DIFF gets ready to load the two files into memory, it checks to see how much space is available in EMS memory, extended memory, and normal memory, then selects the optimal place to store its data. For more details, see the documentation for DIFF in Chapter 13, Section J.

B. TPEMS: EMS Memory Management

This section describes all of the types, variables, constants, procedures and functions found in the interface section of the TPEMS unit.

Constants

```
EmsErrorCode = $0FFFF;
```

Used by several functions in TPEMS to signal an error condition.

Declaration

```
function AllocateEmsPages(NumPages : Word) : Word;
```

Purpose

Allocates the indicated number of pages and returns a handle.

Comments

This function allows you to allocate expanded memory in 16K (16,384 bytes) blocks, or *pages*. You might think of it as being like a combination of GetMem, used to allocate normal memory in Turbo Pascal, and DOS's open file function, which returns a file handle. As with DOS file handles, the handle returned by AllocateEmsPages is your ticket to accessing the memory you've allocated. In particular, it is needed to map a particular page into normal memory (using MapEmsPage) and to deallocate the memory when it is no longer needed (using DeallocateEmsHandle).

AllocateEmsPages returns EmsErrorCode in case of error. The most common cause of error is insufficient memory to fill the request. You can determine the number of pages available at any given time by calling EmsPagesAvail.

Note on handles: Although the standard version of the Expanded Memory Manager assigns handles in the order $FF00, $FE01, $FD02, etc., this order is not guaranteed to be correct for all versions of the EMM driver. At least one EMM driver we are aware of returns handles in the order 0, 1, 2, etc.

Example

```
PagesToAllocate := 4;
if (EmsPagesAvail >= PagesToAllocate) then begin
  MyEmsHandle := AllocateEmsPages(PagesToAllocate);
  if (MyEmsHandle = EmsErrorCode) then
    {error};
end
else
  {insufficient memory};
```

This example shows how to allocate 64K (4 16K pages) of EMS memory using AllocateEmsPages. Notice that EmsPagesAvail is called first to insure that sufficient memory is available.

See Also

DeallocateEmsHandle
EmsPagesAvail
MapEmsPage

DeallocateEmsHandle

Declaration

```
function DeallocateEmsHandle(Handle : Word) : Boolean;
```

Purpose

Deallocates the indicated handle and the memory associated with it.

Comments

This function is the EMS equivalent of Dispose and FreeMem. It allows you to deallocate memory previously allocated with AllocateEmsPages. Note that only the handle returned by AllocateEmsPages is needed, and not the number of pages assigned to it. DeallocateEmsHandle returns False if the handle cannot be deallocated. The most likely cause of error is an invalid handle, or a handle that has already been deallocated.

Very important: Unlike TPEXTMEM, TPEMS does not automatically deallocate the EMS memory you've allocated, primarily because that convenience is not worth the overhead required to provide it. So, any program that uses TPEMS should contain an *exit handler* that will deallocate any EMS memory you've allocated. A simple example is shown below.

Examples

```
MyEmsHandle := AllocateEmsPages(PagesToAllocate);
...
if not DeallocateEmsHandle(MyEmsHandle) then
  {error};
```

This example demonstrates how to deallocate memory previously allocated with AllocateEmsPages:

```
{$F+}
procedure MyExitProc;
begin
  ExitProc := SaveExitProc;
  if not DeallocateEmsHandle(MyEmsHandle) then
    {error if memory already deallocated};
end;
{$F-}
...
SaveExitProc := ExitProc;
ExitProc := @MyExitProc;
```

This example demonstrates a simple exit handler that deallocates all memory assigned to MyEmsHandle when the program ends.

See Also

AllocateEmsPages

Declaration

```
function EmsActiveHandles : Word;
```

Purpose

Returns the number of active EMS handles.

Comments

This function allows you to determine the number of active EMS handles that have been assigned by the Expanded Memory Manager. Like EmsTotalPages, it is useful primarily for displaying status information. EmsActiveHandles returns EmsErrorCode ($FFFF) in case of error. The most likely causes of error are the same as those for EmsStatusOK.

Example

```
WriteLn(LongInt(EmsTotalPages)*16384, ' bytes of EMS memory installed.');
WriteLn(LongInt(EmsPagesAvail)*16384, ' bytes available.');
WriteLn(EmsActiveHandles, ' EMS handles active.');
```

Here the call to EmsActiveHandles allows the program to display the number of handles that are active, as well as the more mundane status information.

See Also

EmsStatusOK EmsTotalPages

Declaration

```
function EmsInstalled : Boolean;
```

Purpose

Returns True if the EMM is installed.

Comments

This function tells you whether or not an Expanded Memory Manager (EMM) is installed. It does *not* tell you if a memory board adhering to the Expanded Memory Specification (EMS) is installed. You should call this routine before using any of the other procedures and functions in the TPEMS unit. In rare cases, you might also need to call EmsVersion to insure that a particular EMS function that your program uses is available.

Example

```
if not EmsInstalled then begin
  WriteLn('This program requires EMS memory. Aborting...');
  Halt(1);
end;
```

In this example, a program that requires EMS memory in order to run aborts if EmsInstalled returns False.

See Also

EmsVersion

EmsPageFramePtr

Declaration

```
function EmsPageFramePtr : Pointer;
```

Purpose

Returns the page frame base address as a pointer to the page frame.

Comments

This function returns a pointer to the base address of the *page frame* used by the EMM. A page frame occupies an entire 64K segment, divided into four 16K pages, numbered 0-3. These four pages are called *physical pages*, as opposed to *logical pages*, because they exist within the 808x's one megabyte address space. Logical pages, by contrast, exist on the expanded memory board, and their numbers are relative to a given handle. If you allocate three pages to a particular handle with AllocateEmsPages, for example, you would refer to them individually as page 0, 1, or 2. In order to access the contents of a particular logical page, moreover, you must *map* the page into one of the four physical pages using MapEmsPage.

Perhaps the easiest way to understand the concept of a page frame is to think in terms of some simple data structures:

```
type EmsPage = array[1..16384] of Byte;
     EmsPageFrame = record
                      Page0, Page1, Page2 : EmsPage;
                      Page3 : array[1..16369] of Byte;
                    end;
     PageFramePtr = ^EmsPageFrame;
```

Notice that, because Turbo Pascal does not allow data types larger than 65521 bytes, Page3 is declared to be smaller than the first three pages. As long as range checking is off, however, we can safely refer to all 16384 elements in the array.

Now, to move 16K of data into the third physical page (#2), for example, you'd say something like

```
MyPageFramePtr := EmsPageFramePtr; {only need to do this once!}
Move(Source, MyPageFramePtr^.Page2[1], 16384);
```

Note that TPEMS does not define these types for you because there's no guarantee that you'll want to treat any given page as an array of byte.

Example

```
var MyPageFramePtr : PageFramePtr;
    MyEmsPagePtr : ^EmsPage;
...
MyPageFramePtr := EmsPageFramePtr;
if not MapEmsPage(MyEmsHandle, 0, 2) then {error};
MyEmsPagePtr := @MyPageFramePtr^.Page2[1];
```

This example shows how you could map the first logical page (#0) associated with MyEmsHandle into the third physical page (#2), and load MyEmsPagePtr with the starting address of the physical page.

See Also

AllocateEmsPages

MapEmsPag

Declaration

```
function EmsPagesAvail : Word;
```

Purpose

Returns number of available pages reported by the Expanded Memory Manager.

Comments

This function allows you to determine how much expanded memory is available for your use. EmsPagesAvail returns the number of 16K (16,384 bytes) pages that are available, or EmsErrorCode in case of error. The most likely causes of error are the same as those for EmsStatusOK. To determine how many pages are actually installed in the system, you can use EmsTotalPages.

Example

```
PagesToAllocate := EmsPagesAvail;
if (PagesToAllocate = EmsErrorCode) then
  {error condition}
else
  if (PagesToAllocate <> 0) then
    MyEmsHandle := AllocateEmsPages(PagesToAllocate);
```

This example shows how to allocate all available EMS memory to your program.

See Also

EmsStatusOK

EmsTotalPages

EmsPagesOwned

Declaration

```
function EmsPagesOwned(Handle : Word) : Word;
```

Purpose

Returns the number of pages owned by Handle.

Comments

This function allows you to determine the number of pages allocated to a particular handle. It will return a value in the range 1-512 if successful, or EmsErrorCode ($FFFF) in case of error. The most likely cause of an error is an invalid handle. EmsPagesOwned is useful primarily for displaying status information.

Example

```
var
  Handle, Pages : Word;
  HandleRec : record
                LowByte, HighByte : Byte;
              end absolute Handle;
begin
  Handle := $FF00;
  repeat
    Pages := EmsPagesOwned(Handle);
    if (Pages <> EmsErrorCode) then
      WriteLn('Handle $', HexW(Handle), ': ', Pages, ' pages');
    Inc(HandleRec.LowByte);
    Dec(HandleRec.HighByte);
  until (Pages = EmsErrorCode);
end;
```

This example demonstrates how to determine how many pages have been allocated to each active EMS handle. (Not guaranteed to work, see the special note on handles in the documentation for AllocateEmsPages.)

See Also

AllocateEmsPages

Declaration

```
function EmsStatusOK : Boolean;
```

Purpose

Returns True if the EMM reports its status as being OK.

Comments

EmsStatusOK allows you to test the expanded memory hardware to insure that it is functional. It returns True if the Expanded Memory Manager reports that its status is OK, False if it does not. The most likely causes of an error are corruption of EMM driver's code or a malfunction in the memory board. Note that EmsStatusOK should be used only after EmsInstalled has been called to establish that the EMM is installed.

Example

```
if not EmsInstalled then begin
  WriteLn('This program requires EMS memory. Aborting...');
  Halt(1);
end;
if not EmsStatusOK then begin
  WriteLn('Malfunction in EMS software or hardware. Aborting...');
  Halt(1);
end;
```

Here EmsInstalled is used to verify that the Expanded Memory Manager is installed, and EmsStatusOK is used to make sure that it is functioning properly.

See Also

EmsInstalled

EmsTotalPages / EmsVersion

Declaration

```
function EmsTotalPages : Word;
```

Purpose

Returns total number of pages of EMS memory, including allocated pages.

Comments

This function returns the number of 16K pages of memory installed, or EmsErrorCode in case of error. The most likely causes of error are the same as those for EmsStatusOK. It is useful primarily for displaying status information.

Example

```
WriteLn(LongInt(EmsTotalPages)*16384, ' bytes of EMS memory installed.');
```

See Also

EmsStatusOK

EmsPagesAvail

Declaration

```
function EmsVersion : Byte;
```

Purpose

Returns a BCD version number of the EMM.

Comments

This function returns a value indicating the version number of the Expanded Memory Manager. The value is in BCD (Binary Coded Decimal) format. To check for 3.2 or greater, for example, you would use 'if EmsVersion >= $32 then...'. EmsVersion returns 0 in case of an error, usually indicating that the EMM is not installed. Note that all of the routines in the TPEMS unit are available in versions 3.0 and higher of the EMM, so EmsVersion should rarely be needed.

Example

```
if EmsVersion < $32 then begin
  WriteLn('Expanded Memory Manager 3.2 or higher required.');
  Halt(1);
end;
```

See Also

EmsInstalled

Declaration

```
function MapEmsPage(Handle, LogicalPage : Word; PhysicalPage : Byte) : Boolean;
```

Purpose

Maps the specified LogicalPage associated with Handle into PhysicalPage (0-3).

Comments

This function is used to make data stored in EMS memory accessible. (*Note:* If you haven't done so already, please read the documentation for EmsPageFramePtr, which defines some of the terms used here.) What it does is to map a particular *logical page* (as defined by the Handle and LogicalPage parameters) into one of the four *physical pages* in the Expanded Memory Manager's page frame. This is what allows you to treat expanded memory as normal memory.

The Handle parameter is the value returned previously by AllocateEmsPages when you allocated your memory. LogicalPage is a number between 0 and *n*-1, where *n* is the number of pages allocated to Handle by AllocateEmsPages. PhysicalPage is a number between 0 and 3, representing one of the four physical pages that make up the page frame.

MapEmsPage returns True if successful, and False if the operation failed. The most likely cause of failure is an invalid handle or page number.

Example

```
type
  EmsPage = array[1..16384] of Byte;
  EmsPageFrame = record
                   Page0, Page1, Page2 : EmsPage;
                   Page3 : array[1..16369] of Byte;

                 end;
  PageFramePtr = ^EmsPageFrame;
var
  MyPageFramePtr : PageFramePtr;
  MyEmsPagePtr : ^EmsPage;
...
MyPageFramePtr := EmsPageFramePtr;
if not MapEmsPage(MyEmsHandle, 0, 2) then
  {error};
MyEmsPagePtr := @MyPageFramePtr^.Page2[1];
```

This example shows how you could map the first logical page (#0) associated with MyEmsHandle into the third physical page (#2), and load MyEmsPagePtr with the starting address of the physical page. See the entry for EmsPageFramePtr for an explanation of why EmsPageFrame is declared the way it is.

See Also

AllocateEmsPages

EmsPageFramePtr

RestoreEmsContext

Declaration

```
function RestoreEmsContext(Handle : Word) : Boolean;
```

Purpose

Restores the mapping context of the EMM driver for the handle specified.

Comments

This function is intended primarily for use in popups that use EMS memory. In conjunction with SaveEmsContext, it allows you to save the mapping context of the Expanded Memory Manager on entry to the popup, and restore it when you exit, so that the interrupted program is not affected. The handle passed as a parameter should be the same one used in a prior call to SaveEmsContext. RestoreEmsContext returns True if restoration is successful.

Example

```
{$F+}
procedure MyPopup;
begin
  if not SaveEmsContext(MyEmsHandle) then
    Exit;
  {-- body of the popup --}
  if not RestoreEmsContext(MyEmsHandle) then
    {error--display warning};
end;
{$F-}
```

This example illustrates how to use RestoreEmsContext in a popup. The popup should save the context on entry, and restore it on exit.

See Also

SaveEmsContext

Declaration

```
function SaveEmsContext(Handle : Word) : Boolean;
```

Purpose

Saves the EMM context for resident programs.

Comments

This function is intended primarily for use in popups that use EMS memory. In conjunction with RestoreEmsContext, it allows you to save the mapping context (i.e., the page mapping registers on the memory board) of the Expanded Memory Manager on entry to the popup, and restore it when you exit, so that the interrupted program is not affected. The handle passed as a parameter must have been allocated previously using a call to AllocateEmsPages. SaveEmsContext returns True if successful.

Example

```
MyEmsHandle := AllocateEmsPages(PagesToAllocate);
{check for EmsErrorCode}
...
{$F+}
procedure MyPopup;
begin
  if not SaveEmsContext(MyEmsHandle) then
    Exit;
  {-- body of the popup --}
  if not RestoreEmsContext(MyEmsHandle) then
    {error--display warning};
end;
{$F-}
```

This example illustrates how to use SaveEmsContext in a popup. The popup should save the context on entry, and restore it on exit. MyEmsHandle is the handle previously assigned to the program with the initial call to AllocateEmsPages.

See Also

AllocateEmsPages

RestoreEmsContext

C. TPEXTMEM: Extended Memory Management

This section describes all of the types, variables, constants, procedures and functions found in the interface section of the TPEXTMEM unit.

Types

```
HugePtr = Pointer;
```

A HugePtr holds a 24-bit linear address. It can be cast to a LongInt (and vice versa) for indexing purposes, and you can test it for a value of nil, just like a normal pointer (which holds a segmented address).

Declaration

```
function ExtMemAvail : Word;
```

Purpose

Returns the number of KB of extended memory that is actually free.

Comments

Much like Turbo's MemAvail, ExtMemAvail allows you to determine how much extended memory is available for your program's use. The number it returns, however, is in kilobytes (1,024 bytes), since that is the unit used by GetExtMem when allocating extended memory. The function returns 0 if DOS 2.x is being used, even if there is actually some extended memory available.

Example

```
if ExtMemAvail >= KBtoAllocate then
  MyHugePtr := GetExtMem(KBtoAllocate)
else begin
  WriteLn('Insufficient extended memory');
  MyHugePtr := nil;
end;
```

In this example, ExtMemAvail is called to see if sufficient memory is available before making an allocation request to GetExtMem.

See Also

ExtMemTotal GetExtMem

Declaration

```
function ExtMemTotal : LongInt;
```

Purpose

Returns total number of bytes of extended memory in the system.

Comments

In contrast to ExtMemAvail, which returns the number of kilobytes of extended memory that is actually available, ExtMemTotal returns the total number of bytes of extended memory installed in the system. It is useful primarily for displaying status information. The function returns 0 if DOS 2.x is being used, even if there is actually some extended memory installed.

Example

```
Write(ExtMemAvail*1024, ' bytes of extended memory available out of ');
WriteLn(ExtMemTotal);
```

See Also

ExtMemAvail

FreeExtMem / GetExtMem

Declaration

```
function FreeExtMem(P : HugePtr) : Boolean;
```

Purpose

Deallocate a block of extended memory.

Comments

FreeExtMem deallocates a block of extended memory allocated with GetExtMem. It works *only* if the block to deallocate is the last block allocated. The function returns False if it was unable to deallocate the memory. Note that it is not necessary to call FreeExtMem at the end of your program, since TPEXTMEM will automatically deallocate any memory you have allocated.

Example

```
MyHugePtr := GetExtMem(KBtoAllocate);
...
if not FreeExtMem(MyHugePtr) then begin
  WriteLn('Unable to dispose of allocated memory'); Exit;
end;
```

GetExtMem is called to allocate extended memory, FreeExtMem to dispose of the memory that was allocated.

See Also

GetExtMem

Declaration

```
function GetExtMem(N : Word) : HugePtr;
```

Purpose

Allocate N KB of extended memory.

Comments

Much like GetMem, GetExtMem lets you allocate a block of extended memory, but it can allocate memory only in 1K (1,024 byte) blocks. Since the function returns nil if there is insufficient memory to fill the request, it is not imperative that you call ExtMemAvail before making the request. A block of extended memory allocated with GetExtMem can later be disposed of using FreeExtMem.

Example

```
MyHugePtr := GetExtMem(KBtoAllocate);
if MyHugePtr = nil then begin
  WriteLn('Unable to allocate extended memory');
  Exit;
end;
```

See Also

ExtMemAvail FreeExtMem

Declaration

```
procedure MarkExtMem(var P : HugePtr);
```

Purpose

Store the location of the next extended memory block in P.

Comments

Like Turbo's Mark, MarkExtMem allows you to store the location of the next free memory block in a variable. You can then allocate multiple blocks of extended memory using GetExtMem, and dispose of all of them at once with ReleaseExtMem. Unlike Turbo's Mark and Release and GetMem and FreeMem, however, TPEXTMEM's MarkExtMem and ReleaseExtMem do not conflict with GetExtMem and FreeExtMem. P will be set to nil if there is no extended memory available.

Example

```
MarkExtMem(ExtMemStart);
HP1 := GetExtMem(KBtoAllocate1);
HP2 := GetExtMem(KBtoAllocate2);
HP3 := GetExtMem(KBtoAllocate3);
...
MemoryReleased := ReleaseExtMem(ExtMemStart);
```

In this example, MarkExtMem is called before allocating several blocks of extended memory with GetExtMem, and ReleaseExtMem is called to dispose of it when it is no longer needed.

See Also

GetExtMem

ReleaseExtMem

MoveExtMem

Declaration

```
procedure MoveExtMem(Src, Dest : HugePtr; WCount : Word);
```

Purpose

Move WCount words of extended memory from Src to Dest.

Comments

Much like Turbo's Move, MoveExtMem allows you to move blocks of memory between any two locations in memory. That is, you can use it to transfer data from extended memory to normal memory (or vice versa), or between two regions of extended memory. Unlike Move, however, MoveExtMem does not allow you to move data in one-byte units--a *very important* caveat to keep in mind when using the routines in TPEXTMEM.

Note: When using Move with normal pointers, you must dereference the pointers:

```
Move(PointA^, PointB^, BytesToMove);
```

When using MoveExtMem with HugePtr's, however, you must *not* dereference them (using the '^' symbol). Also when transferring data from normal to extended memory and back again, the address in normal memory must first be converted from a segmented address (pointer) to a linear address (huge pointer, or HugePtr) using PtrToHuge. For example,

```
MoveExtMem(PtrToHuge(@MyArray[1]), MyHugePtr, 512);
```

would move 512 words (1,024 bytes) from MyArray[1] to the address pointed to by MyHugePtr.

Examples

```
MoveExtMem(PtrToHuge(MyBufferPtr)^, MyHugePtr^, WordsToMove);
```

Incorrect! The huge pointers must not be dereferenced.

```
MoveExtMem(PtrToHuge(MyBufferPtr), MyHugePtr, WordsToMove);
```

Correct. This would move WordsToMove words of data from the address in normal memory pointed to by MyBufferPtr to the address pointed to by MyHugePtr.

See Also

PtrToHuge

Declaration

```
function PtrToHuge(P : Pointer) : HugePtr;
```

Purpose

Converts a pointer to a huge pointer.

Comments

PtrToHuge converts a regular pointer representing a segmented address to a huge pointer representing a linear address. Among other things, PtrToHuge is needed to move data between normal and extended memory using MoveExtMem. Note that the pointer passed to PtrToHuge must be in the range $0:$0 to $FFFF:$000F.

Example

```
MoveExtMem(PtrToHuge(MyBufferPtr), MyHugePtr, WordsToMove);
```

Declaration

```
procedure ReleaseExtMem(P : HugePtr);
```

Purpose

Release all extended memory above P.

Comments

Much like Release, ReleaseExtMem lets you dispose of all memory blocks above a given point in memory, a point whose location was stored in a HugePtr variable using MarkExtMem. You can thus allocate multiple blocks of extended memory with GetExtMem, and dispose of all of them at once with ReleaseExtMem. Unlike Turbo's Mark-Release and GetMem-FreeMem, TPEXTMEM's MarkExtMem-ReleaseExtMem pair do not conflict with GetExtMem-FreeExtMem. Note also that it is not necessary to call ReleaseExtMem at the end of your program, since TPEXTMEM will automatically dispose of any memory you have allocated.

Very important: The P parameter *must* have been set by MarkExtMem. It cannot be a HugePtr returned by GetExtMem.

Example

```
MarkExtMem(ExtMemStart);
HP1 := GetExtMem(KBtoAllocate1);
HP2 := GetExtMem(KBtoAllocate2);
...
ReleaseExtMem(ExtMemStart);
```

MarkExtMem is called before allocating several blocks of extended memory with GetExtMem, and ReleaseExtMem releases them when they are no longer needed. Calling ReleaseExtMem would be unnecessary if the program were about to end.

See Also

MarkExtMem

9. Macros

A. Overview

The term *macro* is widely used in a number of ways. Many general purpose programs, notably spreadsheets, incorporate a *macro language* that allows you to tailor them for particular tasks. Similarly, some compilers and assemblers allow you to assign a symbolic name to a sequence of instructions; Turbo Pascal's "inline procedures" can be considered *macros* in this sense of the word. More frequently, however, the term refers to *keyboard macros*, a facility that allows you to make a single keystroke do the work of many. Most commercial text editors and word processors include such a facility, and now Turbo Professional allows you to incorporate this powerful feature into your own applications.

The TPMACRO unit contains all the low-level routines you should need to do this. They allow you to define macros, assign names to them, record them, play them back, save them to a file for future use, and retrieve them when they're needed again.

Once defined, keyboard macros can be played back in one of two ways. If macro processing is enabled (by a call to MacrosOn), a macro will automatically be played back any time that the corresponding key is pressed. Alternatively, you can force a particular macro to be played back by calling either StartMacro or StartMacroByKey. In either case, the task of playing back the macro is performed by TPMACRO's hidden macro engine, which can also record macros for you.

One important restriction is worth noting here. TPMACRO does not allow "nested macros." For example, if one macro (call it macro A) has been assigned to <F1>, a second one (call it macro B) to <F2>, and macro A contains <F2>, macro B will *not* be played back as part of macro A. There are two main reasons for this restriction. First, it avoids the possibility of an infinite loop, such as might occur if macro A in our example contained <F2> and macro B contained <F1>. Second, it allows TPMACRO to be used to remap the keyboard (e.g., you could use it to reverse the functions of <F1> and <F3>).

Although the ability to record and play back keyboard macros will be sufficient for some applications, in many cases you will also need a way to edit these macros after they've been defined. For this reason, Turbo Professional contains a related unit, TPMACED, that implements a full-featured macro editor. This specialized editor allows you to easily insert, delete, and change the keys in a macro, which are represented as strings of one or more characters. The Escape key, for example, is

displayed as "<Esc>", and the Enter key as "<Enter>", so you don't need to know anything about scan codes in order to use it.

Turbo Professional's keyboard macros are stored in variably sized records of type MacroRec. The first field in the record, NumKeys, is a word indicating the number of keys in the macro. The second field, KeyArray, is an array of words with a character code in the low byte and a scan code in the high byte, the same format used by the BIOS. The end of the array is marked by a word with the value EndOfMacro ($FFFF, or -1). You might think of this format as being like a cross between Turbo-style strings, which start with a length byte, and ASCIIZ strings, which use null characters (#0) to mark the ends of strings. Each macro is associated with a key, such as <F1>, and an optional name, such as 'Help'. By default, you are limited to 100 (MaxMacros) such macros of up to 1000 keystrokes (MaxKeysInMacro) each, but because Turbo Professional comes with complete source code, you can raise or lower these limits whenever you want.

A program that uses TPMACRO in protected mode must use CRT or TPCRT, or the macro keys will not be played back. When CRT or TPCRT are not used, keyboard input is obtained by making a DOS int $21 call. In this situation, the protected mode int $16 handler installed by TPMACRO never receives control.

For a comprehensive example of how to use Turbo Professional's keyboard macros and macro editor, see the demonstration program SMACS.PAS. SMACS is a memory resident macro processor that includes most of the features found in commercial programs such as Borland's SuperKey. It comes with a file of sample macros, TURBO5.MAC, that you may find handy while using the Turbo Pascal editor. For further details, see the documentation for SMACS in Chapter 13, Section A.

B. TPMACRO: Keyboard Macros

This section describes all of the types, variables, constants, procedures and functions found in the interface section of the TPMACRO unit.

Constants

```
EndOfMacro = $FFFF;
```

Used to mark the end of a macro. Also used to mark unused entries in the DefinedKeys array.

```
Int16Handle = 12;
```

Number assigned to the int $16 handler.

```
MacroFileID : LongInt = $04001087;
```

Macro file identification values. These values are stored at the beginning of any valid macro file to insure its validity.

```
MaxKeysInMacro = 1000;
```

The maximum number of keys in a macro. If you want to increase this limit, please keep in mind that a MacroRec contains an array[1..MaxKeysInMacro] of word (see below), and that both TPMACRO and TPMACED declare a global variable of type MacroRec. In other words, the absolute limit for this constant is slightly over 32000, or 16000 if TPMACED is used. The practical limit is much lower.

```
MaxMacros = 100;
```

The maximum number of macros that TPMACRO can handle at one time.

```
PlaybackDelay : Word = 0;
```

Number of milliseconds to delay before playing back a particular keystroke in a macro.

Types

```
MacroRec = record
             NumKeys : Word;
             KeyArray : array[1..MaxKeysInMacro] of Word;
             Overflow : Word;
           end;
```

A macro. NumKeys is the number of keys in the macro, not counting the last, which is always EndOfMacro ($0FFFF). KeyArray contains the actual keystrokes. Overflow holds the end of macro marker when KeyArray is completely full.

```
MacroRecPtr = ^MacroRec;
```

Pointer to a macro record.

```
StringPointer = ^String;
```

Pointer to a string.

Variables

```
AuxKeyLength : array[0..255] of Byte;
```

Length of names for auxiliary keys (ones with extended scan codes).

```
DefinedKeys : array[1..MaxMacros] of Word;
```

Table of defined macro keys. Unused entries contain EndOfMacro.

```
MacroInProgress : Boolean;
```

If True, a macro is currently being played back.

```
MacroNames : array[1..MaxMacros] of Pointer;
```

Array of pointers to macro names.

```
MacroPointers : array[1..MaxMacros] of Pointer;
```

Array of pointers to defined macros.

```
MacroRecording : Boolean;
```

If True, macro recording is enabled.

```
MacrosAreOn : Boolean;
```

If True, macro processing is enabled.

```
RegKeyLength : array[0..255] of Byte;
```

Length of names for regular keys.

```
ScrapMacro : MacroRec;
```

Used by macro engine when recording keystrokes.

```
ScrapMacroKey : Word;
```

Can be used to store the key associated with ScrapMacro.

```
ScrapMacroName : string;
```

Can be used to store the name associated with ScrapMacro.

Declaration

```
function AllocateMacro(Key : Word; var Macro; var Name : string) : Byte;
```

Purpose

Allocates a macro on the heap.

Comments

This routine defines a macro and allocates space for both the macro and its name (if any). If there is already a macro assigned to Key, both the old macro and the old name are deleted and their memory deallocated. AllocateMacro returns one of the following values:

0	Macro allocated
1	Macro table full
2	Out of heap space
3	Macro was blank

Note that a return value of 3 is not necessarily an error code.

Example

```
var
  TempMacroKey : Word;
  TempMacro : MacroRec;
const
  NullString : string[1] = '';
begin
  {--get TempMacroKey and TempMacro--}
  case AllocateMacro(TempMacroKey, TempMacro, NullString) of
    {0 or 3 are OK, 1 and 2 are errors}
     1 : WriteLn('Macro table full');
     2 : WriteLn('Out of memory');
    else WriteLn('Macro saved');
  end;
end;
```

See Also

AssignMacroName
DeallocateMacro
DefineMacro

AssignMacroName / CharToMacro

Declaration

```
function AssignMacroName(Key : Word; var Name : string) : Boolean;
```

Purpose

Assign a Name to a macro.

Comments

This routine allows you to change a macro name or assign a name to a macro that doesn't yet have one. It deletes the existing name assigned to the macro, if any, and returns False if there is insufficient memory for Name or if the macro associated with Key is not found.

Example

```
var Name : string;
begin
  Write('Enter name for macro: ');
  ReadLn(Name);
  if not AssignMacroName(ScrapMacroKey, Name) then
    WriteLn('Insufficient memory or macro not found');
end;
```

Assign a new name to the macro associated with ScrapMacroKey.

See Also

AllocateMacro

Declaration

```
function CharToMacro(MacroChar : Char) : Word;
```

Purpose

Converts a character to a keyboard code/character representation.

Comments

This function allows you to easily convert ASCII characters to key codes. SMACS, for example, used this routine in its screen reader to translate characters read off the screen into a macro. See the StringToMacro routine in SMACS.PAS for a good example of what CharToMacro can be used for.

See Also

KeyToString

Declaration

```
procedure ClearMacros;
```

Purpose

Clears out the current macros.

Comments

ClearMacros irreversibly deletes and deallocates all macros and macro names.

See Also

DeallocateMacro DefineMacro

Declaration

```
function DeallocateMacro(Key : Word) : Boolean;
```

Purpose

If there is a macro associated with Key, erase it and deallocate its memory.

Comments

This routine is the counterpart to AllocateMacro. If Key is associated with a macro, it deletes both the macro and the corresponding macro name (if any), and deallocates their memory. DeallocateMacro returns False if Key is not associated with a macro.

Example

```
if not DeallocateMacro(ScrapMacroKey) then {error} ;
```

Get rid of any existing macro associated with ScrapMacroKey.

See Also

AllocateMacro DefineMacro

DefineMacro / FindMacroIndex

Declaration

```
function DefineMacro(Key : Word; var Macro) : Boolean;
```

Purpose

Define a new macro or delete an existing one.

Comments

This low level routine lets you execute the main functions of AllocateMacro and DeallocateMacro, without their automatic allocation/deallocation of dynamic memory. It is intended primarily for internal use.

To define a macro: Key is the key to associate with Macro. DefineMacro will return False if the table is full or the macro is blank.

To delete a macro: Key is the key to delete and Macro must be a blank macro (NumKeys = 0). DefineMacro will return False if the macro was not found.

See the AllocateMacro and DeallocateMacro routines in TPMACRO.PAS for examples.

See Also

AllocateMacro

DeallocateMacro

Declaration

```
function FindMacroIndex(Key : Word) : Word;
```

Purpose

Find the array index for the macro assigned to Key.

Comments

This routine returns the array index for the macro corresponding to Key, or 0 if not found. This index can in turn be used to access elements in the MacroPointers, MacroNames, and DefinedKeys arrays. It is intended primarily for internal use, but you may need it to create routines that cannot be written using the higher level routines in TPMACRO. You might also want to use it as a fast alternative to MacroAddress, when checking to see if a particular key has already been defined. SMACS does this in several places, for example.

Example

```
Index := FindMacroIndex(MacroKey);
if Index <> 0 then WriteLn('Name: ', MacroNames[Index]^);
```

See Also

MacroAddress

Declaration

```
function InitMacros : Boolean;
```

Purpose

Initializes macro engine.

Comments

InitMacros activates the macro engine. You might wish to deactivate it (using RemoveMacros) before executing a DOS shell, for example, then reactivate it with InitMacros. After a call to InitMacros, no macros are defined and both macro processing and macro recording are turned off. InitMacros returns False if int $16 cannot be taken over. This can happen only if your program has already used the ISR handle used by the macro engine (Int16Handle). You do not need to call InitMacros at the start of your program, since that is done automatically.

Example

```
RemoveMacros;
Code := ExecDos('', True, nil);
if not InitMacros then {error};
```

See Also

RemoveMacros

Declaration

```
procedure KeyToString(Key : Word; var S : string; var Special : Boolean);
```

Purpose

Returns a string (S) representing a Key.

Comments

This routine can be used, for example, when you want to display a list of defined macro keys. Special is False if a simple character (like <A>) is being returned, and True when a key with an extended scan code (like <F1>) is being represented.

Example

```
var I, Key : Word;
    S : string; Special : Boolean;
begin
  for I := 1 to MaxMacros do begin
    Key := DefinedKeys[I];
    if Key <> EndOfMacro then begin
      KeyToString(Key, S, Special); WriteLn(S);
    end;
  end;
end;
```

Displays a list of all keys that have macros attached to them.

See Also

CharToMacro

MacroAddress / MacroCount

Declaration

```
function MacroAddress(Key : Word) : Pointer;
```

Purpose

Returns a pointer to the macro for a key, or nil if the key is not defined.

Comments

This routine allows you to determine whether or not a particular key has a macro associated with it. You might want to call this routine, for example, before calling StartMacroByKey or to see if you should ask the user to verify that an existing macro should be overwritten.

Example

```
var SavedKey : Word;
begin
  Write('Press key for macro to be played back later: ');
  MacrosOff;
  SavedKey := ReadKeyWord;
  MacrosOn;
  WriteLn;
  if MacroAddress(SavedKey) = nil then begin
    WriteLn('Macro not defined'); Exit;
  end;
  ...
  StartMacroByKey(SavedKey);
end;
```

See Also

StartMacroByKey

Declaration

```
function MacroCount : Word;
```

Purpose

Returns the number of macros that have been defined.

Comments

The result is a number between 0 and MaxMacros (100).

Example

```
WriteLn(MacroCount, ' macros defined');
```

MacroRecordingOff / MacroRecordingOn

Declaration

```
procedure MacroRecordingOff;
```

Purpose

Turns macro recording off.

Comments

Calling this routine simply tells the macro engine to stop saving keystrokes in ScrapMacro. It is your responsibility to assign this macro to a key and allocate space for it (using AllocateMacro, most likely).

Examples

See the PopupRecordingOff routine in SMACS.PAS for an example.

See Also

AllocateMacro

MacroRecordingOn

Declaration

```
procedure MacroRecordingOn;
```

Purpose

Turns macro recording on.

Comments

Calling this routine tells the macro engine to start saving all keystrokes entered by the user in ScrapMacro. It will continue saving keystrokes until recording is turned off with a call to MacroRecordingOff.

Examples

See the PopupRecordingOn routine in SMACS.PAS for an example.

See Also

MacroRecordingOff

MacrosOff / MacrosOn

Declaration

```
procedure MacrosOff;
```

Purpose

Turns macro processing off.

Comments

This routine disables macro processing. SMACS calls this routine, for example, before prompting for the key associated with a macro to be edited. After the user has pressed a key, macro processing is reenabled with a call to MacrosOn.

Example

```
MacrosOff;
W := ReadKeyWord;
MacrosOn;
```

Turn macro processing off, read from the keyboard, turn it back on.

See Also

MacrosOn

Declaration

```
procedure MacrosOn;
```

Purpose

Turns macro processing on.

Comments

This routine enables macro processing. Although some programs will simply need to call this routine once at the beginning of the program, in other cases you may need to temporarily turn macro processing off. SMACS, for example, disables macro processing before prompting for the key associated with a macro to be edited, then reenables it by calling this routine.

Example

```
MacrosOff;
W := ReadKeyWord;
MacrosOn;
```

Turn macro processing off, read from the keyboard, turn it back on.

See Also

MacrosOff

Declaration

```
function ReadMacroFile(FName : String; Merge : Boolean) : Byte;
```

Purpose

Read a macro file.

Comments

This routine reads the contents of a macro file into memory. If Merge is False, any existing macros are deleted before the file is read. If True, existing macros that do not conflict with new ones are left in memory. ReadMacroFile returns 0 if successful, $FF if not a valid macro file, $FE if it ran out of memory, else IoResult. *Note:* Since this routine does not check for DOS critical errors, we recommend that you not use it in TSRs.

Examples

```
if ReadMacros('MYMACROS.1', False) <> 0 then {error};
```

Existing macros overwritten.

```
if ReadMacros('MYMACROS.2', True) <> 0 then {error};
```

Existing macros not overwritten.

See Also

WriteMacroFile

Declaration

```
procedure RemoveMacros;
```

Purpose

Restores all vectors associated with macros.

Comments

This routine disables the macro engine. You might wish to do this before executing a DOS shell, for example. The macro engine can be reactivated by calling InitMacros.

Example

```
RemoveMacros;
Code := ExecDos('', True, nil);
if not InitMacros then {error};
```

See Also

InitMacros

SetWaitCount / StartMacro

Declaration

```
procedure SetWaitCount(Count : Byte);
```

Purpose

Set a wait count that is used between each character of a macro.

Comments

This routine sets the number of times a KeyPressed function will receive a False response between characters of a macro. The default is 0, so KeyPressed will return True every time, until the macro is fully played back. Use a larger value when playing back macros over a program that seems to "eat" the keys.

Declaration

```
procedure StartMacro(Macro : Pointer);
```

Purpose

Starts execution of the specified macro.

Comments

This routine lets you start any macro, even one that hasn't been officially defined (with DefineMacro or AllocateMacro). If a macro is already in progress it will immediately be replaced by the macro passed.

Examples

```
var
  MyMacroRec : MacroRec;
  MyMacroPtr : MacroRecPtr;
begin
  {--define MyMacroRec--}
  MyMacroPtr := @MyMacroRec;
  StartMacro(MyMacroPtr^);
end;
```

Incorrect! The '^' in MyMacroPtr^ is not needed.

```
StartMacro(MyMacroPtr);
```

Correct.

```
StartMacro(@MyMacroRec);
```

Also correct.

See Also

StartMacroByKey

Declaration

```
procedure StartMacroByKey(Key : Word);
```

Purpose

Starts execution of the macro attached to key, if any.

Comments

Unlike StartMacro, this routine allows you to play back a macro when all you know is the key associated with it. If a macro is already in progress, it will immediately be replaced by the specified macro.

Examples

```
const
  F1Key = $3B00;
...
StartMacroByKey(F1Key);
```

Play back the macro associated with <F1>, if any.

```
var
  SavedKey : Word;
begin
  Write('Press key for macro to be played back later: ');
  MacrosOff;
  SavedKey := ReadKeyWord;
  MacrosOn;
  ...
  StartMacroByKey(SavedKey);
end;
```

Get a macro key, do something, play back the macro.

See Also

StartMacro

StringToMacro / StringToScrapMacro

Declaration

```
procedure StringToMacro(S : string; MRP : MacroRecPtr; MaxKeys : Word);
```

Purpose

Initialize a macro from a string.

Comments

This routine turns string (S) into a macro. MRP should point to the macro record used to store the macro. MaxKeys is the maximum number of keys that MRP^ can hold.

Example

```
S := 'this is a test of StringToMacro'^M;
StringToMacro(S, @ScrapMacro, MaxKeysInMacro);
```

Converts S to a macro and stores it in ScrapMacro.

See Also

CharToMacro

StringToScrapMacro

Declaration

```
procedure StringToScrapMacro(S : string);
```

Purpose

Initialize ScrapMacro from a string.

Comments

This routine forms a convenient "shell" around StringToMacro. It converts S to a macro and stores it in the global variable ScrapMacro. Note that this routine must not be used while macro recording is on, since ScrapMacro is used to store a macro that is being recorded.

Example

```
S := 'this is a test of StringToScrapMacro'^M;
StringToScrapMacro(S);
```

Converts S to a macro and stores it in ScrapMacro.

See Also

StringToMacro

Declaration

```
function WriteMacroFile(FName : String) : Byte;
```

Purpose

Write a macro file.

Comments

This routine saves the current macros to a file. It returns 0 if successful, else IoResult. *Note:* Since this routine does not check for DOS critical errors, we recommend that you not use it in TSRs.

Example

```
if WriteMacroFile('MYMACROS.3') <> 0 then {error};
```

See Also

ReadMacroFile

C. TPMACED: Macro Editor

This section describes all of the types, variables, constants, procedures and functions found in the interface section of the TPMACED unit.

Constants

```
NonMacroKey = $FFFF;
```

If this value is passed to EditKeys as the Key parameter, Key is not translated into a string and displayed while editing.

Variables

```
MacEdKeypressedPtr : Pointer;
```

Pointer to a keypressed routine. MacEdKeypressedPtr points to a function of the form:

```
{$F+}
function MyKeyPressed : Boolean;
begin
  {return True if a key has been pressed, otherwise False}
end;
{$F-}
```

By default the pointer is set to '@TPCRT.KeyPressed'. Although it is generally safe to perform background tasks within the routine pointed to by MacEdKeypressedPtr, the routine must give up control as soon as a key is pressed *or* the keyboard flags change. (The status of the keyboard flags can be obtained by calling the KbdFlags function in TPCRT.)

```
TempMacro : MacroRec;
```

Temporary macro used for editing. Since this macro is used only while editing, you may treat it as a scrap macro in most situations, as is done in SMACS.PAS.

Declaration

```
function EditKeys(Key : Word; var Macro : MacroRec;
                  MinCol, MinRow, MaxCol, MaxRow, Dim, Bright : Byte;
                  var Modified : Boolean) : MacroRecPtr;
```

Purpose

Edit the Macro associated with Key.

Comments

EditKeys implements a full-featured macro editor. The editing window is defined by the MinCol, MinRow, MaxCol, and MaxRow parameters. The window should probably be at least 40 columns wide and five rows high, preferably more. Since the window scrolls vertically, you don't need to worry about macros that are too long to be displayed in their entirety.

The Dim and Bright parameters determine the video attributes used to draw the window border (Bright) and the keys in the macro (normal keys are displayed in the Dim attribute; keys that must be represented with a string, like <Esc>, <Enter>, or <F1> are displayed in the bright attribute).

Key is the key associated with the macro, and normally it is displayed at the top left corner of the window. If you don't want it displayed, pass NonMacroKey as the Key parameter.

Macro, a variable of type MacroRec, is never actually modified by EditKeys. Instead, EditKeys makes a copy of it, on which the actual editing is performed, and returns a pointer to the copy. If no edits were made to the Macro, the Modified parameter is set to False, otherwise to True.

The following commands are available in the macro editor:

<Enter>
Stop editing.

<Esc>
Restore original macro and stop editing.

<Left>
Cursor left one key.

<Right>
Cursor right one key.

<Up>
Cursor up one line.

<Down>
Cursor down one line.

<Ins>
Toggle insert mode.

<Del>
Delete key at cursor.

<Bksp>
Delete previous key.

<CtrlBksp>
Delete all keys in macro.

<ScrollLock>
Toggle between command mode and literal mode. In literal mode, pressing any of the keys listed above will insert that key into the macro, rather than executing the command.

Example

```
const
  NullString : string[1] = '';
var
  P, MyMacroPtr : MacroRecPtr;
  Modified : Boolean;
begin
  MacrosOff;
  Write('Macro key: ');
  ScrapMacroKey := ReadKeyWord;
  MacrosOn;

  {look for existing macro}
  P := MacroAddress(ScrapMacroKey);
  if P = nil then begin
    {macro not found--create blank macro to edit}
    ScrapMacro.NumKeys := 0;
    ScrapMacro.KeyArray[1] := EndOfMacro;
    P := @ScrapMacro;
  end;

  {edit the macro}
  MyMacroPtr := EditKeys(ScrapMacroKey, P^, 1, 1,
    ScreenWidth, ScreenHeight, $07, $0F, Modified);
  ClrScr;

  {if edits were made, allocate the macro (gets rid of old one)}
  if Modified then
    case AllocateMacro(ScrapMacroKey, MyMacroPtr^, NullString) of
      1..2 : WriteLn('Out of memory');
    end;
end;
```

This example demonstrates how to set up a full screen macro editor using EditKeys. For another example, see the EditMacro routine in SMACS.PAS.

10. Large Arrays

A. Overview

Although Turbo Pascal provides a large code model, and has always offered access to all 640K of DOS memory via the heap, and now even provides access to 16MB of memory in DOS protected mode applications, it still does not allow individual data arrays to exceed 64K bytes in size. Turbo Professional supplies three units--TPRARRAY, TPVARRAY, and TPEARRAY--that help you circumvent this limitation. Turbo Professional's large arrays can be one- or two-dimensional, and array elements can be of any type.

The three large array units allow creation of arrays of virtually any size. TPRARRAY stores array data in normal RAM, on the Pascal heap, and as a result it is limited to arrays that will fit into available memory. TPVARRAY is a virtual model, storing the bulk of an array in a file, and paging portions of the array into memory on demand. Arrays managed by TPVARRAY are limited in size only by disk space and the maximum file size of the operating system, typically 32 megabytes. TPEARRAY stores the array in Lotus/Intel/Microsoft expanded memory (EMS). These arrays are limited by EMS version 3 to 8 megabytes, or more recently by EMS version 4 to 32 megabytes.

TPRARRAY can be used in DOS protected mode with Borland Pascal 7.0, leading to a maximum array size of 16MB. TPVARRAY can also be used in protected mode. TPEARRAY is for DOS real mode only.

The calling interface for all three units is almost identical. In all cases, identical procedure and function names are used to access the supported operations: array allocation, initialization, and deallocation; setting and returning elements; and storing and loading arrays from disk files. Minor variations in the parameter lists exist only as required by the implementation of each unit. Because the calling conventions are so similar, the array units can be used interchangeably. Using more than one of the units in a given program is also possible by taking advantage of Turbo Pascal's unit name override mechanism. For example:

```
TPRARRAY.MakeA(A, 100, 100, SizeOf(Word));
TPEARRAY.MakeA(A, 100, 100, SizeOf(Word));
TPVARRAY.MakeA(A, 100, 100, SizeOf(Word), 'ARRAY.DAT', 50000);
```

Each of the units is fully dynamic--that is, the dimensions and data type of an array are specified at runtime rather than compile time. As a result, a single copy of the large array code will manage multiple data arrays of differing sizes and types in a

single program. The disadvantage of this approach is that the large array unit cannot define a function that returns the value of an array element, since the type of the element is not known until runtime. Instead, each unit exports a procedure that returns array elements as untyped VAR parameters. Your application can easily define a shell function that calls this procedure and returns the value it returns as a function result of the appropriate type.

Arrays managed by Turbo Professional's large array units are always zero-based; the minimum index on each axis of an array is always zero. When a new array is allocated, you will specify the total number of rows and columns of the array. When the array is accessed, the highest array index will be one less than the actual number of rows or columns.

Data types exceeding 64K bytes inherently take longer to access than smaller objects when using the Intel 8086 family architecture. Furthermore, dynamically allocated structures take longer to access than statically addressed ones. For each access of an array element, Turbo Professional's routines must calculate a page (segment) and offset within the page where the data actually will be found. This calculation is performed in assembly language to attain the best possible speed. The virtual and EMS arrays require an additional operation to assure that the desired page is loaded into normal memory.

The following table provides some benchmark results for reading and writing elements in each array model. Turbo Pascal 4.0's performance for a small array is also included for comparison. Measurements were taken on an 8 MHz 80286 system. Results are reported as the number of microseconds to read or write a single array element of the specified type.

Element	Turbo	RAM	Virtual	EMS
Byte	8	88	125	135
Word	8	89	126	137
Real	16	93	129	141
String[80]	96	153	190	235

The virtual model results do not include time for swapping from disk, nor do the EMS results include time for swapping page windows. Disk swap times will vary significantly depending on the hardware. EMS page swaps take 100-150 microseconds. TPVARRAY supports a user-definable number of in-RAM pages, and TPEARRAY supports two independent EMS page windows, in order to minimize the amount of swapping required.

Common Mistakes

Virtually all of the technical support calls that we receive about these three units are the result of a single common mistake: forgetting that arrays are always zero-based. In the most common case, the user has declared a one-dimensional array and tried to refer to the *n*'th element as row *n*, column 1, rather than row *n*-1, column 0. Please verify that you have not made such a mistake before calling our technical support line for assistance.

B. TPRARRAY: Arrays in Normal RAM

This section describes all of the types, variables, constants, procedures and functions found in the interface section of the TPRARRAY unit.

Constants

```
WriteError : Boolean = True;
HaltOnError : Boolean = True;
RangeCheck : Boolean = False;
```

These typed constants determine how TPRARRAY handles errors. If WriteError is True, a descriptive error message is written to the standard output device when an array error occurs. If HaltOnError is True, execution halts after the error message is written. If RangeCheck is True, all accesses to array elements are checked for illegal values and an error is reported if an access is out of range. Although range checking slows the performance of TPRARRAY, it should be the first thing you turn on if an application is not working correctly. Any application using TPRARRAY can change these typed constants at runtime. If HaltOnError is False, the application should call the function ErrorA to determine whether an array error has occurred. See the description of ErrorA for more details.

Types

```
TpArray = Pointer;
```

RAM-based arrays consume only the space of a pointer in the segment where they are declared. The TPRARRAY unit takes the approach of Modula-2 by exporting an "opaque" type for the array. In actuality, a TpArray type points to a record structure which provides further information about the implementation details. A program using the unit should not require access to these details, though, and therefore they are not interfaced. Refer to the TpArrayInternal type declaration in the implementation section of TPRARRAY for further details.

```
InitType = (ExactInit, FastInit);
```

This enumerated type defines initialization options for arrays. Passing the ExactInit value to ClearA causes that routine to initialize each array element to an exact value of the proper type. Passing FastInit causes ClearA to set every byte of the array to the first byte of the initialization value. As its name implies, the second option is much faster. FastInit is useful for initializing numeric arrays to zero or string arrays to empty strings.

Declaration

```
procedure ClearA(A : TpArray; var Value; Initialize : InitType);
```

Purpose

Initialize all array values to Value.

Comments

Notice that Value is a VAR parameter, so you will need to assign the initialization value to a variable of the appropriate type and then pass it to ClearA. Initialize can be either ExactInit or FastInit (see the entry for the InitType). If FastInit is used, ClearA initializes the entire array to contain the first byte in Value. If ExactInit is used, ClearA assigns the exact Value to each array element.

Examples

```
ClearA(A, 0, FastInit)
```

Incorrect! The Value parameter must first be assigned to a variable.

```
Value := 0;
ClearA(A, Value, FastInit)
```

This is the correct usage.

Declaration

```
procedure DisposeA(var A : TpArray);
```

Purpose

Deallocate array.

Comments

TPRARRAY stores the bulk of each large array on the Pascal heap. Whenever you call MakeA, an appropriate amount of memory is allocated. By calling DisposeA, you will deallocate all the space previously allocated for an array, thus freeing it for another use.

You need *not* call DisposeA at the end of an application because Turbo's entire heap will be freed automatically at that time.

Example

```
MakeA(A, 200, 200, SizeOf(Word));
DisposeA(A);
```

The MakeA statement will allocate approximately 80000 bytes of heap space (200*200*2, plus about 20 bytes of overhead). DisposeA deallocates all of the space allocated by MakeA.

See Also

MakeA

ErrorA

Declaration

```
function ErrorA : Word;
```

Purpose

Return the last error number and reset the error variable.

Comments

When the HaltOnError control variable is set to False (see the Declarations section), TPRARRAY will return to the calling application after errors occur. It is the application's responsibility to call the ErrorA function to determine whether an error has occurred.

ErrorA works in a manner similar to that of Turbo's IoResult. ErrorA returns a value of zero if the previous operation was successful, and a specific non-zero value if an error occurred. After each call to ErrorA, the internal error storage variable is cleared.

Error values returned by ErrorA are as follows:

1 Row index out of range
(reported during a call to RetA or SetA only when RangeCheck is True)

2 Column index out of range
(reported during a call to RetA or SetA only when RangeCheck is True)

3 Invalid element size
(reported if ElementSize is zero during a call to MakeA or MakeAPrimitive)

4 Invalid row or column limit
(reported if the total number of rows or columns is zero during a call to MakeA or MakeAPrimitive)

5 Invalid page size
(reported if RowsInPage*ColsInPage exceeds 64K during a call to MakeAPrimitive)

6 Invalid page index size
(reported if 4*PagesDown*PagesAcross exceeds 64K during a call to MakeAPrimitive)

7 Insufficient memory to allocate array
(reported during a call to MakeA or MakeAPrimitive)

8 File creation error
(reported during a call to StoreA)

9 File write error
(reported during a call to StoreA)

10 Error while closing file
(reported during a call to StoreA or LoadA)

11 File not found
(reported during a call to LoadA)

12 File read error
(reported during a call to LoadA)

17 Invalid file format
(reported during a call to LoadA)

Example

```
RetA(A, 20, 1000, V);
E := ErrorA;
if E <> 0 then begin
  writeln('TPRARRAY error ',E);
  halt;
end;
```

Note that E is first stored to a variable. If ErrorA were called twice during the example, the second value would report zero.

See Also

Declarations

LoadA

Declaration

```
procedure LoadA(var A : TpArray; FileName : string);
```

Purpose

Allocate and read array from disk.

Comments

LoadA first attempts to open the file specified in FileName. It is the calling program's responsibility to assure the file exists. The file must have been created with an earlier call to StoreA.

From the file, LoadA reads a header that describes the size and type of the array. Based on this information, LoadA calls MakeAPrimitive to allocate heap space for array storage. Then LoadA reads the data from the disk file into heap storage and finally closes the disk file.

Note that the file formats used by TPRARRAY, TPEARRAY and TPVARRAY are not interchangeable. The LoadA and StoreA routines form a matched pair within each library.

Examples

```
LoadA(A, 'STATS.ARR');
LoadA(A, 'C:\DATA\101587.DAT');
```

See Also

ErrorA

Declaration

```
procedure LoadVA(var A : TpArray; FileName : string);
```

Purpose

Allocate and read array from disk in VARRAY format.

Comments

Use this routine to load arrays previously initialized by StoreVA in this unit or by StoreA in the TPVARRAY unit. The TPVARRAY file format can be used as a method of interchanging arrays between TPRARRAY and TPVARRAY. This may be useful when a single program uses TPRARRAY or TPVARRAY calls depending on the amount of available memory detected at runtime.

Before calling LoadVA, assure that FileName is an existing file.

Example

```
uses
  TpRarray;
var
  A : TpArray;
begin
  LoadVA(A, 'TEST.ARR');
end.
```

Loads an array previously stored by TPRARRAY.StoreVA or TPVARRAY.StoreA.

See Also

StoreA (TPVARRAY) StoreVA

MakeA

Declaration

```
procedure MakeA(var A : TpArray; Rows : Word; Cols : Word; ElementSize : Word);
```

Purpose

Allocate RAM-based array the easy way.

Comments

Call this routine to dynamically create a new array. The parameter A must contain a pointer variable of type TpArray. Rows and Cols specify the total number of elements along each axis of the array. ElementSize specifies the number of bytes in each element of the array. The easiest way to determine ElementSize is to use Turbo's built-in SizeOf operator (see example below).

MakeA automatically determines a page structure for the array. Its criteria for page structure are: each page is limited to 64K bytes, each page forms an identical rectangular subsection of the total array, and the overhead for pointers to array pages plus wasted page space at the edges is to be minimized. After MakeA determines such a page structure, it calls MakeAPrimitive which actually allocates the array space and initializes the necessary header fields.

Note that MakeA performs no initialization of the array elements. Without further initialization (perhaps by a call to ClearA), array elements will contain random values.

If MakeA finds an error in the array specification, or if sufficient memory is not available, it will return an error code in the ErrorA function, or halt, depending on the error handling control variables.

Examples

```
MakeA(A, 100, 100, SizeOf(Word));
```

Allocates a 100x100 array of type Word.

```
MakeA(A, 1000, 0, SizeOf(String));
```

Incorrect! The total number of columns must be at least one.

```
MakeA(A, 1000, 1, SizeOf(String));
```

Allocates a one-dimensional array with 1000 rows, each capable of holding a 255 character string.

See Also

ErrorA

MakeAPrimitive

Declaration

```
procedure MakeAPrimitive(var A : TpArray; RowsInPage, ColsInPage,
                         PagesDown, PagesAcross, ElementSize : Word);
```

Purpose

Allocate array memory, with full control over page structure.

Comments

MakeAPrimitive is, as its name implies, a lower level alternative to the MakeA procedure. Whereas MakeA automatically determines an array's internal page structure, MakeAPrimitive gives the calling program full control over the way an array is organized.

Any one data structure in Turbo Pascal is limited to 64K bytes. To manage large arrays, TPRARRAY breaks a structure larger than 64K into a group of identical "pages," each of which is smaller than 64K. TPRARRAY manages these pages by initializing a list of pointers, one of which points to each page. Then, whenever an array reference is made to a particular row and column, TPRARRAY computes the page number and page offset where the data actually resides.

MakeA divides a given array automatically into a page structure that it considers optimal. It may be possible, however, that you can devise a page structure that wastes less heap space than MakeA does. In that case you should call MakeAPrimitive rather than MakeA.

Based on the parameters passed to MakeAPrimitive, the total number of array rows is equal to PagesDown * RowsInPage, and the number of columns is equal to PagesAcross * ColsInPage.

MakeAPrimitive will return with an error if it detects a problem with the array specification, or if sufficient heap space is not available.

Examples

```
MakeAPrimitive(A, 200, 200, 1, 1, SizeOf(Word));
```

Incorrect! This requests a page size of 80000 bytes (200*200*2), which is beyond the 64K limit.

```
MakeAPrimitive(A, 100, 100, 2, 2, SizeOf(Word));
```

Allocates a 200x200 array of type Word.

```
MakeA(A, 100, 0, 10, 1, SizeOf(String));
```

Incorrect! The total number of columns must be at least one.

```
MakeA(A, 100, 1, 10, 1, SizeOf(String));
```

Allocates a one-dimensional array with 1000 rows, each capable of holding a 255 character string.

See Also

ErrorA

MakeA

RetA

Declaration

```
procedure RetA(A : TpArray; R, C : Word; var Value);
```

Purpose

Return value of A[R, C].

Comments

All arrays managed by TPRARRAY are *zero-based*. By this, we mean that the lowest index on either axis of the array is zero. By the same token, the highest index on either axis is one less than the total number of rows or columns in the array.

RetA returns the current value of the array element at position R, C. You may specify the name of the array variable in the first parameter to RetA. Since TPRARRAY doesn't know the type of the element (only its size), it returns the answer in an untyped VAR parameter. Your calling routine should pass a variable of the appropriate type in this position.

If range checking is enabled, and the row or column value is outside of the array bounds, RetA will halt or return an error value via ErrorA, depending on the settings of the error control variables.

In many cases it is desirable to have a function (rather than a procedure) to return the values of array elements. The TPRARRAY unit cannot define such a function because at compile time the data type for the array is not defined. Fortunately, it is very easy for your application to define one or more such functions to return the specific data types of your application. Here's an example:

```
function GetA(A : TpArray; R, C : Word) : MyDataType;
var X : MyDataType;
begin
  RetA(A, R, C, X); GetA := X;
end;
```

This approach will work whenever MyDataType is a type for which functions may be defined. These types include Byte, Word, LongInt, Pointer, Real and String. Unfortunately, record types may not be returned as function results. You could define a function which returned any simple field of a record, though:

```
function GetField(A : TpArray; R, C : Word) : MyFieldType;
var X : MyRecordType;
begin
  RetA(A, R, C, X); GetField := X.MyField;
end;
```

Examples

```
RetA(A, 0, 0, W);
```

Returns the very first element in array A to the variable W

```
RetA(A, TotalRows-1, TotalCols-1, W);
```

Returns the very last element from array A

See Also

ErrorA SetA

Declaration

```
procedure SetA(A : TpArray; R, C : Word; var Value);
```

Purpose

Set array element A[R, C] to value.

Comments

All arrays managed by TPRARRAY are *zero-based*. By this, we mean that the lowest index on either axis of the array is zero. By the same token, the highest index on either axis is one less than the total number of rows or columns in the array.

SetA sets the current value of the array element at position R, C. You may specify the name of the array variable in the first parameter to SetA. Since TPRARRAY doesn't know the type of the element (only its size), the new value is passed as an untyped VAR parameter. Your calling routine should pass a variable of the appropriate type in this position.

If range checking is enabled, and the row or column value is outside of the array bounds, SetA will halt or return an error value via ErrorA, depending on the settings of the error control variables.

In many cases it is desirable to pass new array elements by using constants or expressions. TPRARRAY cannot support this directly because an array's data type is not known at compile time, but you can easily define one or more such procedures to set the specific data types of your application. Here's an example:

```
procedure SetVal(A : TpArray; R, C : Word; V : MyDataType);
begin
  SetA(A, R, C, V);
end;
```

SetVal can then be called with constants or expressions in the position for V.

Note that TPRARRAY provides no means to change a single field within a record type array element. Again, it is easy for your application to define such a routine:

```
function SetField(A : TpArray; R, C : Word; V : MyFieldType);
var X : MyRecordType;
begin
  RetA(A, R, C, X);
  X.MyField := V;
  SetA(A, R, C, X);
end;
```

Examples

```
SetA(A, 0, 0, W);
```

Sets the very first element in array A to the variable W

```
SetA(A, TotalRows-1, TotalCols-1, W);
```

Sets the very last element from array A

See Also

ErrorA

RetA

StoreA

Declaration

```
procedure StoreA(A : TpArray; FileName : string);
```

Purpose

Write array to disk.

Comments

StoreA creates a new file as specified by the parameter FileName. StoreA does not check to see whether this file already exists--it is the calling program's responsibility to do so if desired.

StoreA first writes a header to the file, describing the array's dimensions and type. Then StoreA writes each of the array's pages to disk, and finally closes the file. StoreA does not deallocate the array's image on the heap. If storing an array is the last need for it in a particular program, call DisposeA to deallocate the RAM space after the array is written.

The file size of the array file should be only slightly larger than the actual array size, given by the formula ArraySize = TotalRows * TotalCols * ElementSize.

StoreA will return an error if it is unable to create, write to, or close the array file.

Examples

```
StoreA(A, 'C:\AUTOEXEC.BAT');
```

Watch out! StoreA will overwrite an existing file without warning.

```
StoreA(A, 'STATS.DAT');
```

Stores array A to STATS.DAT.

See Also

LoadA

Declaration

```
procedure StoreVA(A : TpArray; FileName : string);
```

Purpose

Write array to disk in VARRAY format.

Comments

Use this routine to store an array initialized by TPRARRAY in a format that can later be read and used by TPVARRAY. This might be desirable in two situations: a small program initializes constant arrays, later to be used by a larger program (constrained for memory) which must therefore use TPVARRAY; or a single program can use either TPRARRAY or TPVARRAY calls depending on how much memory it detects at runtime.

A is a large array previously initialized by TPRARRAY calls. FileName is the name of a file where output will be written. If FileName already exists, StoreVA will overwrite it without warning.

Example

```
uses
  TpRarray;
const
  R : Real = 0.0;
var
  A : TpArray;
begin
  MakeA(A, 100, 100, SizeOf(Real));
  ClearA(A, R, FastInit);
  StoreVA(A, 'TEST.ARR');
end.
```

Initializes a 100x100 real array with zeros. The array file TEST.ARR can later be read into a program that uses TPVARRAY with the following statement:

```
LoadA(A, 'TEST.ARR', RamBytesForData);
```

See Also

LoadA (TPVARRAY) LoadVA

C. TPEARRAY: Arrays in EMS Memory

This section describes all of the types, variables, constants, procedures and functions found in the interface section of the TPEARRAY unit.

Constants

```
WriteError : Boolean = True;
HaltOnError : Boolean = True;
RangeCheck : Boolean = False;
```

These typed constants determine how TPEARRAY handles errors. If WriteError is True, a descriptive error message is written to the standard output device when an array error occurs. If HaltOnError is True, execution halts after the error message is written. If RangeCheck is True, all accesses to array elements are checked for illegal values and an error is reported if an access is out of range. Although range checking slows the performance of TPEARRAY, it should be the first thing you turn on if an application is not working correctly. Any application using TPEARRAY can change these typed constants at runtime. If HaltOnError is False, the application should call the function ErrorA to determine whether an array error has occurred. See the description of ErrorA for more details.

Types

```
TpArray = Pointer;
```

Expanded memory arrays consume only the space of a pointer in the segment where they are declared. The TPEARRAY unit takes the approach of Modula-2 by exporting an "opaque" type for the array. In actuality, a TpArray type points to a record structure which provides further information about the implementation details. A program using the unit should not require access to these details, though, and therefore they are not interfaced. Refer to the TpArrayInternal type declaration in the implementation section of TPEARRAY for further details.

```
InitType = (ExactInit, FastInit);
```

This enumerated type defines initialization options for new arrays. Passing the ExactInit value to ClearA causes that routine to initialize each array element to an exact value of the proper type. Passing FastInit causes ClearA to set every byte of the array to the first byte of the initialization value. As its name implies, the second option is much faster. FastInit is useful for initializing numeric arrays to zero or string arrays to empty strings.

Variables

```
EmsAvailable : Boolean;
```

The initialization block of the TPEARRAY unit sets this variable True or False depending on whether it senses an installed EMS driver. If EmsAvailable is False, your application cannot make use of the TPEARRAY routines.

Declaration

```
procedure ClearA(A : TpArray; var Value; Initialize : InitType);
```

Purpose

Initialize all array values to Value.

Comments

Notice that Value is a VAR parameter, so you will need to assign the initialization value to a variable of the appropriate type and then pass it to ClearA. Initialize can be either ExactInit or FastInit (see the entry for the InitType). If FastInit is used, ClearA initializes the entire array to contain the first byte in Value. If ExactInit is used, ClearA assigns the exact Value to each array element.

Examples

```
ClearA(A, 0, FastInit)
```

Incorrect! The Value parameter must first be assigned to a variable.

```
Value := 0;
ClearA(A, Value, FastInit)
```

This is the correct usage for a fast initialization with a zero byte.

Declaration

```
procedure DisposeA(var A : TpArray);
```

Purpose

Deallocate array.

Comments

Although TPEARRAY stores a small array header (about 10 bytes) on the heap, the bulk of an expanded memory array is stored in expanded memory. By calling DisposeA, you will deallocate both the heap space and the expanded memory space for the array. Unlike the other array models, it is very important that your application call the DisposeA routine for every allocated array before the application exits to DOS. Otherwise the expanded memory will not be freed for later use.

Example

```
MakeA(A, 200, 200, SizeOf(Word));
DisposeA(A);
```

The MakeA statement will allocate approximately 80000 bytes of expanded memory space (200*200*2) plus about 10 bytes of normal heap space. DisposeA deallocates all of the space allocated by MakeA.

See Also

MakeA

ErrorA

Declaration

```
function ErrorA : Word;
```

Purpose

Return the last error number and reset the error variable.

Comments

When the HaltOnError control variable is set to False (see the Declarations section), TPEARRAY will return to the calling application after errors occur. It is the application's responsibility to call the ErrorA function to determine whether an error has occurred.

ErrorA works in a manner similar to that of Turbo's IoResult. ErrorA returns a value of zero if the previous operation was successful, and a specific non-zero value if an error occurred. After each call to ErrorA, the internal error storage variable is cleared.

Error values returned by ErrorA are as follows:

1 Row index out of range
(reported during a call to RetA or SetA only when RangeCheck is True)

2 Column index out of range
(reported during a call to RetA or SetA only when RangeCheck is True)

3 Invalid element size
(reported if ElementSize is zero during a call to MakeA)

4 Invalid row or column limit
(reported if the total number of rows or columns is zero during a call to MakeA)

7 Insufficient memory to allocate array
(reported if insufficient heap space is available during a call to MakeA)

8 File creation error
(reported during a call to StoreA)

9 File write error
(reported during a call to StoreA)

10 Error while closing file
(reported during a call to StoreA or LoadA)

11 File not found
(reported during a call to LoadA)

12 File read error
(reported during a call to LoadA)

13 EMS page mapping error
(reported when an attempt to remap the EMS page window fails)

14 EMS driver and/or hardware not installed
(reported after an attempt to use TPEARRAY when EmsAvailable is false)

15 Insufficient EMS memory to allocate array
(reported if insufficient expanded memory space is available during a call to MakeA)

16 Error deallocating EMS memory
(reported during a call to DisposeA)

17 Invalid file format
(reported during a call to LoadA)

Example

```
RetA(A, 20, 1000; V);
E := ErrorA;
if E <> 0 then begin
  writeln('TPEARRAY error ',E);
  halt;
end;
```

Note that E is first stored to a variable. If ErrorA were called twice during the example, the second value would report zero.

See Also

Declarations

LoadA

Declaration

```
procedure LoadA(var A : TpArray; FileName : string);
```

Purpose

Allocate and read array from disk.

Comments

LoadA first attempts to open the file specified in FileName. It is the calling program's responsibility to assure that the file exists. The file must have been created with an earlier call to StoreA.

From the file, LoadA reads a header that describes the size and type of the array. Based on this information, LoadA calls MakeA to allocate memory space for array storage. Then LoadA reads the data from the disk file into the allocated space and finally closes the disk file.

Note that the file formats used by TPEARRAY, TPEARRAY, and TPVARRAY are not interchangeable. The LoadA and StoreA routines form a matched pair within each library.

Examples

```
LoadA(A, 'STATS.ARR');
LoadA(A, 'C:\DATA\101587.DAT');
```

See Also

ErrorA

Declaration

```
procedure MakeA(var A : TpArray; Rows : Word; Cols : Word; ElementSize : Word);
```

Purpose

Allocate array the easy way.

Comments

Call this routine to dynamically create a new array. The parameter A must contain a pointer variable of type TpArray. Rows and Cols specify the total number of elements along each axis of the array.

ElementSize specifies the number of bytes in each element of the array. The easiest way to determine ElementSize is to use Turbo's built-in SizeOf operator (see example below). For expanded memory arrays, the maximum size of any one element is 16K bytes. This limitation is imposed by the 16K page window size of the Expanded Memory Specification, and by the desire to have at least two independent windows onto expanded memory.

EMS memory imposes a fixed 16K byte page size on the organization of the array. As such, the actual usage of expanded memory will always be rounded up to the next multiple of 16K bytes.

Note that MakeA performs no initialization of the array elements. Without further initialization (perhaps by a call to ClearA), array elements will contain random values.

If MakeA finds an error in the array specification, or if sufficient memory is not available, it will return an error code in the ErrorA function, or halt, depending on the error handling control variables.

Examples

```
MakeA(A, 100, 100, SizeOf(Word));
```

Allocates a 100x100 array of type Word.

```
MakeA(A, 1000, 0, SizeOf(String));
```

Incorrect! The total number of columns must be at least one.

```
MakeA(A, 1000, 1, SizeOf(String));
```

Allocates a one-dimensional array with 1000 rows, each capable of holding a 255 character string.

See Also

ErrorA

RetA

Declaration

```
procedure RetA(A : TpArray; R, C : Word; var Value);
```

Purpose

Return value of A[R,C].

Comments

All arrays managed by TPEARRAY are *zero-based*. By this, we mean that the lowest index on either axis of the array is zero. By the same token, the highest index on either axis is one less than the total number of rows or columns in the array.

RetA returns the current value of the array A at position R, C. Since TPEARRAY doesn't know the type of the element (only its size), it returns the answer in an untyped VAR parameter. Your calling routine should pass a variable of the appropriate type in this position.

If range checking is enabled, and the row or column value is outside of the array bounds, RetA will halt or return an error value via ErrorA, depending on the settings of the error control variables.

In many cases it is desirable to have a function (rather than a procedure) to return the values of array elements. The TPEARRAY unit cannot define such a function because at compile time the data type for the array is not defined. Fortunately, it is very easy for your application to define one or more such functions to return the specific data types of your application. Here's an example:

```
function GetA(A : TpArray; R, C : Word) : MyDataType;
var X : MyDataType;
begin
  RetA(A, R, C, X);
  GetA := X;
end;
```

This approach will work whenever MyDataType is a type for which functions may be defined. These types include Byte, Word, LongInt, Pointer, Real and String. Unfortunately, record types may not be returned as function results. See the entry for RetA in the TPRARRAY section for a way to work around this limitation.

Examples

```
RetA(A, 0, 0, W);
```

Returns the very first element in array A to the variable W

```
RetA(A, TotalRows-1, TotalCols-1, W);
```

Returns the very last element from array A

See Also

ErrorA

SetA

Declaration

```
procedure SetA(A : TpArray; R, C : Word; var Value);
```

Purpose

Set array element A[R,C] to value.

Comments

All arrays managed by TPEARRAY are *zero-based.* By this, we mean that the lowest index on either axis of the array is zero. By the same token, the highest index on either axis is one less than the total number of rows or columns in the array.

SetA sets the current value of array A at position R, C. Since TPEARRAY doesn't know the type of the element (only its size), the new value is passed as an untyped VAR parameter. Your calling routine should pass a variable of the appropriate type in this position.

If range checking is enabled, and the row or column value is outside of the array bounds, SetA will halt or return an error value via ErrorA, depending on the settings of the error control variables.

In many cases it is desirable to pass new array elements by using constants or expressions. The TPEARRAY unit cannot directly support this because an array's data type is not known at compile time. Fortunately, it is very easy for your application to define one or more such procedures to set the specific data types of your application. Here's an example:

```
procedure SetVal(A : TpArray; R, C : Word; V : MyDataType);
begin
  SetA(A, R, C, V);
end;
```

SetVal can then be called with constants or expressions in the position for V.

Note also that TPEARRAY provides no direct means to change a single field within a record type array element. See the entry for SetA in the TPRARRAY section for a way to work around this limitation.

Examples

```
SetA(A, 0, 0, W);
```

Sets the very first element in array A to the value contained in W

```
SetA(A, TotalRows-1, TotalCols-1, W);
```

Sets the very last element of array A

See Also

ErrorA RetA

StoreA

Declaration

```
procedure StoreA(A : TpArray; FileName : string);
```

Purpose

Write array to disk.

Comments

StoreA creates a new file as specified by the parameter FileName. StoreA does not check to see whether this file already exists--it is the calling program's responsibility to do so if desired.

StoreA first writes a header to the file, describing the array's dimensions and type. Then StoreA writes each of the array's pages to disk, and finally closes the file. StoreA does not deallocate the array's image on the heap. If storing an array is the last need for it in a particular program, call DisposeA to deallocate the RAM space after the array is written.

The file size of the array file should be only slightly larger than the actual array size, given by the formula ArraySize = TotalRows * TotalCols * ElementSize. Note, however, that the last 16K of the array is written in full, independent of how much real data it contains.

StoreA will return an error if it is unable to create, write to, or close the array file.

Examples

```
StoreA(A, 'C:\AUTOEXEC.BAT');
```

Watch out! StoreA will overwrite an existing file without warning.

```
StoreA(A, 'STATS.DAT');
```

Stores array A to STATS.DAT.

See Also

LoadA

D. TPVARRAY: Arrays in Virtual Memory

This section describes all of the types, variables, constants, procedures and functions found in the interface section of the TPVARRAY unit.

Constants

```
WriteError : Boolean = True;
HaltOnError : Boolean = True;
RangeCheck : Boolean = False;
```

These typed constants determine how TPVARRAY handles errors. If WriteError is True, a descriptive error message is written to the standard output device when an array error occurs. If HaltOnError is True, execution halts after the error message is written. If RangeCheck is True, all accesses to array elements are checked for illegal values and an error is reported if an access is out of range. Although range checking slows the performance of TPVARRAY, you should turn it on if an application is not working correctly. If HaltOnError is False, the application should call ErrorA to determine whether an array error occurred.

```
KeepDiskCurrent : Boolean = False;
```

KeepDiskCurrent determines how often TPVARRAY updates the physical disk where the bulk of a virtual array is stored. If KeepDiskCurrent is False, pages are written only when they are forced from memory. The page index is not written, nor the DOS disk buffers flushed, until the array is closed or a FlushA command is executed. If KeepDiskCurrent is True, the page index is updated whenever the RAM image of the index is changed, and DOS disk buffers are flushed after every disk write. Setting KeepDiskCurrent True can result in a significant slowdown for arrays that do much paging. We recommend judicious use of the FlushA command in combination with KeepDiskCurrent = False for production code.

Types

```
TpArray = Pointer;
```

RAM-based arrays consume only the space of a pointer in the segment where they are declared. The TPVARRAY unit takes the approach of Modula-2 by exporting an "opaque" type for the array. In actuality, a TpArray type points to a record structure which provides further information about the implementation details. A program using the unit should not require access to these details, though, and therefore they are not interfaced. Refer to the TpArrayInternal type declaration in the implementation section of TPVARRAY for further details.

```
InitType = (ExactInit, FastInit);
```

This enumerated type defines initialization options for new arrays. Passing the ExactInit value to ClearA causes that routine to initialize each array element to an exact value of the proper type. Passing FastInit causes ClearA to set every byte of the array to the first byte of the initialization value. As its name implies, the second option is much faster. FastInit is useful for initializing numeric arrays to zero or string arrays to empty strings.

ClearA / DisposeA

Declaration

```
procedure ClearA(A : TpArray; var Value; Initialize : InitType);
```

Purpose

Initialize all array values to Value.

Comments

Notice that Value is a VAR parameter, so you will need to assign the initialization value to a variable of the appropriate type and then pass it to ClearA. Initialize can be either ExactInit or FastInit (see the entry for the InitType). If FastInit is used, ClearA initializes the entire array to contain the first byte in Value. If ExactInit is used, ClearA assigns the exact Value to each array element.

Examples

```
ClearA(A, 0, FastInit)
```

Incorrect! The Value parameter must first be assigned to a variable.

```
Value := 0;
ClearA(A, Value, FastInit)
```

This is the correct usage.

Declaration

```
procedure DisposeA(var A : TpArray; DeleteFile : boolean);
```

Purpose

Deallocate array in memory, and delete array file if requested.

Comments

TPVARRAY stores a portion of each large array on the Pascal heap. Whenever you call MakeA or MakeAPrimitive, an appropriate amount of memory is allocated. By calling DisposeA, you will deallocate all the space previously allocated for an array, thus freeing it for another use. If the DeleteFile parameter to DisposeA is True, DisposeA will also delete the file used for virtual storage of the array. If DeleteFile is false, the file is updated and closed but not deleted. You need *not* call DisposeA at the end of an application because Turbo's entire heap will be freed automatically at that time.

Example

```
MakeA(A, 200, 200, SizeOf(Word), 'TEST.ARR', 80000);
DisposeA(A, True);
```

MakeA will allocate approximately 80000 bytes of heap space. DisposeA will deallocate all 80000 bytes, and it will delete TEST.ARR.

See Also

MakeA

Declaration

```
function ErrorA : Word;
```

Purpose

Return the last error number and reset the error variable.

Comments

When the HaltOnError control variable is set to False (see the Declarations section), TPVARRAY will return to the calling application even after errors occur. It is the application's responsibility to call the ErrorA function to determine whether an error has occurred.

ErrorA works in a manner similar to that of Turbo's IoResult. ErrorA returns a value of zero if the previous operation was successful, and a specific non-zero value if an error occurred. After each call to ErrorA, the internal error storage variable is cleared.

Error values returned by ErrorA are as follows:

1 Row index out of range
(reported during a call to RetA or SetA only when RangeCheck is True)

2 Column index out of range
(reported during a call to RetA or SetA only when RangeCheck is True)

3 Invalid element size
(reported if ElementSize is zero during a call to MakeA or MakeAPrimitive)

4 Invalid row or column limit
(reported if the total number of rows or columns is zero during a call to MakeA or MakeAPrimitive)

5 Invalid page size
(reported if RowsInPage*ColsInPage exceeds 64K during a call to MakeAPrimitive)

6 Invalid page index size
(reported if 4*PagesDown*PagesAcross exceeds 64K during a call to MakeAPrimitive)

7 Insufficient memory to allocate array
(reported during a call to MakeA or MakeAPrimitive)

8 File creation error
(reported during a call to StoreA, MakeA, or MakeAPrimitive)

9 File write error
(reported during a call to StoreA or when paging occurs)

10 Error while closing file
(reported during a call to StoreA, LoadA, or DisposeA)

11 File not found
(reported during a call to LoadA)

12 File read error
(reported during a call to LoadA)

13 Number of in-RAM pages less than 1
(reported during a call to MakeA, MakeAPrimitive, or LoadA when the allotted heap space is not large enough to hold at least one array page)

17 Invalid file format
(reported during a call to LoadA)

ErrorA

Example

```
RetA(A, 20, 1000, V);
E := ErrorA;
if E <> 0 then begin
  writeln('TPVARRAY error ',E);
  halt;
end;
```

Note that E is first stored to a variable. If ErrorA were called twice during the example, the second value would report zero.

See Also

Declarations

Declaration

```
procedure FlushA(A : TpArray);
```

Purpose

Update the disk image of the array.

Comments

FlushA first writes out any *dirty pages* (those whose RAM image differs from the disk image), and then updates the *page index* (which describes the file offset of each page). Finally, it flushes the DOS buffers to assure that the physical disk is fully updated. The virtual array disk file remains open.

Example

```
FlushA(A);
```

Updates the disk image of array A

See Also

References

LoadA

Declaration

```
procedure LoadA(var A : TpArray; FileName : string; RamBytesForData : LongInt);
```

Purpose

Allocate and read array from disk.

Comments

LoadA first attempts to open the file specified in FileName. It is the calling program's responsibility to assure that the file exists. The file is in a special binary format--it must have been created by an earlier call to StoreA.

From the file, LoadA reads a header that describes the size and type of the array. Based on this information, LoadA calls MakeAPrimitive to allocate heap space for array storage. Then LoadA reads the page index from the disk file into heap storage. The disk file remains open for further access to the virtual data. No data is initially read into memory--instead data is paged in on demand.

RamBytesForData specifies how much heap space should be reserved for data pages. It must be at least as large as the size of one data page. The value need not be the same as the value specified when the array was first made. Beyond the data pages, LoadA uses additional heap space to hold indices used in managing the array: memory overhead is approximately (6*TotalArrayPages)+200 bytes. For example, consider an array totaling 300K bytes organized as 5 pages of 60K bytes each. Overhead would consume about 230 bytes of heap space, plus about 60K bytes for each array page kept in RAM.

Note that the file formats used by TPRARRAY, TPEARRAY and TPVARRAY are not interchangeable. The LoadA and StoreA routines form a matched pair within each library.

Examples

```
LoadA(A, 'STATS.ARR', MaxAvail div 2);
```

Allocates half of the available heap space to data storage for the array STATS.ARR.

```
LoadA(A, 'C:\DATA\101587.DAT', 1000);
```

Watch out! If the page size of this array exceeds 1000 bytes (which is very likely), this statement will generate an error.

See Also

ErrorA
MakeA
MakeAPrimitive
StoreA

Declaration

```
procedure MakeA(var A : TpArray; Rows : Word; Cols : Word; ElementSize : Word;
                FileName : string; RamBytesForData : LongInt);
```

Purpose

Allocate a virtual array the easy way.

Comments

Call this routine to dynamically create a new virtual array. The parameter A must contain a pointer variable of type TpArray. Rows and Cols specify the total number of elements along each axis of the array. ElementSize specifies the number of bytes in each element of the array. The easiest way to determine ElementSize is to use Turbo's built-in SizeOf operator (see example below).

FileName specifies the pathname of a file which will hold the array's virtual data. Disk space must exist for the actual data size of the array plus a small overhead (see below for details). MakeA does not check to see whether a file of the same name already exists. If one does, it will be overwritten.

RamBytesForData specifies how much RAM space should be devoted to a paging area between disk and your program. As you allot more RAM space, less paging will occur and array performance will improve. You must allocate enough RAM space for at least one page of the array. If desired, you may allocate RAM space to hold the entire array in memory, in which case the disk will not be accessed except as a backup device.

MakeA automatically determines a page structure for the array. Its criteria for page structure are: each page is limited to 64K bytes, each page forms an identical rectangular subsection of the total array, and the overhead for array management plus wasted page space at the edges is to be minimized. These criteria lead MakeA to minimize the number of pages, making each page as close to 64K bytes as is consistent with the other rules. After MakeA determines such a page structure, it calls MakeAPrimitive which actually allocates the array space and initializes the necessary header fields.

If MakeA determines that the array should be broken into TotalPages pages, and that RamPages pages will be kept in memory at any one time, then memory and disk space consumed as overhead can be computed as follows:

```
RamOverhead = (6*TotalPages)+(4*RamPages)+200 bytes
RamDataSpace = (PageSize+20)*RamPages bytes
DiskOverhead = (4*TotalPages)+20 bytes
DiskDataSpace = PageSize*TotalPages
```

Note that MakeA performs no initialization of the array elements. Without further initialization (perhaps by a call to ClearA), array elements will contain random values.

If MakeA finds an error in the array specification, or if sufficient memory is not available, it will return an error code in the ErrorA function, or halt, depending on the error handling control variables.

MakeA

Examples

```
MakeA(A, 1000, 1000, SizeOf(Word), 'BIG.ARR', MemAvail div 2);
```

Allocates a 1000x1000 array (1 million elements) of type Word using the file BIG.ARR for virtual storage and allocating half of the available heap space for a paging area. MakeA would divide this array into 32 pages of 62500 bytes each. Assuming that available heap space allowed for 4 pages in RAM simultaneously, then the space factors defined above would be:

RamOverhead = 412 bytes
RamDataSpace = 250080 bytes
DiskOverhead = 148 bytes
DiskDataSpace = 2000000 bytes

```
MakeA(A, 1000, 0, SizeOf(String), 'BIG.ARR', 100000);
```

Incorrect! The total number of columns must be at least one.

```
MakeA(A, 1000, 1, SizeOf(String), 'BIG.ARR', 100000);
```

Allocates a one-dimensional array with 1000 rows, each capable of holding a 255 character string, with virtual storage in BIG.ARR and 100000 bytes of heap space used for data storage.

See Also

ErrorA

MakeAPrimitive

Declaration

```
procedure MakeAPrimitive(var A : TpArray; RowsInPage, ColsInPage,
                         PagesDown, PagesAcross, ElementSize : Word;
                         FileName : string; RamBytesForData : LongInt);
```

Purpose

Allocate array memory, with full control over page structure.

Comments

MakeAPrimitive is, as its name implies, a lower level alternative to MakeA. Whereas MakeA automatically determines an array's internal page structure, MakeAPrimitive lets the calling program control the way an array is organized.

MakeA divides a given array automatically into a page structure that it considers optimal. In the case of virtual arrays, though, MakeA is missing an important piece of information which might allow it to optimize array performance as well as memory space. Because the disk device used for paging is so much slower than CPU speed or memory accesses, the number of disk accesses for array paging should be minimized to maintain good array performance. If an array is accessed primarily in row order (that is, the row number is incremented rapidly, and the column number is incremented less often) then each page should hold the maximum possible number of rows. The converse holds if an array is accessed primarily in column order. MakeA might choose the wrong internal organization. With MakeAPrimitive, you can assure that a desired organization is used.

Based on the parameters passed to MakeAPrimitive, the total number of rows is PagesDown*RowsInPage, and the number of columns is PagesAcross*ColsInPage. The total number of pages is PagesDown*PagesAcross, and the size of each page is RowsInPage*ColsInPage*ElementSize. Refer to the MakeA section to determine overhead factors for an array, and for further details regarding the array FileName and RamBytesForData.

MakeAPrimitive will return with an error if it detects a problem with the array specification, or if sufficient heap space is not available.

Examples

```
MakeAPrimitive(A, 200, 200, 1, 1, SizeOf(Word), 'DATA.ARR', 100000);
```

Incorrect! This requests a page size of 80000 bytes (200*200*2), which is beyond the 64K limit allowed by Turbo Pascal.

```
MakeAPrimitive(A, 100, 100, 2, 2, SizeOf(Word), 'DATA.ARR', 100000);
```

Allocates a 200x200 array of type Word. If 100000 bytes of heap space are available, the entire array will be kept in memory with no paging required.

```
MakeAPrimitive(A, 100, 1, 10, 1, SizeOf(String), 'DATA.ARR', 100000);
```

Allocates a one-dimensional array with 1000 rows, each capable of holding a 255 character string. Three pages of 10 total will be kept in RAM.

See Also

ErrorA MakeA

RetA

Declaration

```
procedure RetA(A : TpArray; R, C : Word; var Value);
```

Purpose

Return value of A[R, C].

Comments

All arrays managed by TPVARRAY are *zero-based*. By this, we mean that the lowest index on either axis of the array is zero. By the same token, the highest index on either axis is one less than the total number of rows or columns in the array.

RetA returns the current value of the element at position R, C. You specify the name of the array variable in the first parameter to RetA. Since TPVARRAY doesn't know the type of the element (only its size), it returns the answer in an untyped VAR parameter. Your calling routine should pass a variable of the appropriate type in this position.

If range checking is enabled, and the row or column value is outside of the array bounds, RetA will halt or return an error value via ErrorA, depending on the settings of the error control variables.

In many cases it is desirable to have a function (rather than a procedure) to return the values of array elements. The TPVARRAY unit cannot define such a function because at compile time the data type for the array is not defined. Fortunately, it is very easy for your application to define one or more such functions to return the specific data types of your application. Here's an example:

```
function GetA(A : TpArray; R, C : Word) : MyDataType;
var
  X : MyDataType;
begin
  RetA(A, R, C, X);
  GetA := X;
end;
```

This approach will work whenever MyDataType is a type for which functions may be defined. These types include Byte, Word, LongInt, Pointer, Real and String. Unfortunately, record types may not be returned as function results. See the entry for RetA in the TPRARRAY section for a way to work around this limitation.

Examples

```
RetA(A, 0, 0, W);
```

Returns the very first element in array A to the variable W

```
RetA(A, TotalRows-1, TotalCols-1, W);
```

Returns the very last element from array A

See Also

ErrorA

SetA

Declaration

```
procedure SetA(A : TpArray; R, C : Word; var Value);
```

Purpose

Set array element A[R, C] to value.

Comments

All arrays managed by TPVARRAY are *zero-based*. By this, we mean that the lowest index on either axis of the array is zero. By the same token, the highest index on either axis is one less than the total number of rows or columns in the array.

SetA sets the current value of the element at position R, C. You specify the name of the array variable in the first parameter to SetA. Since TPVARRAY doesn't know the type of the element (only its size), the new value is passed as an untyped VAR parameter. Your calling routine should pass a variable of the appropriate type in this position.

If range checking is enabled, and the row or column value is outside of the array bounds, SetA will halt or return an error value via ErrorA, depending on the settings of the error control variables.

In many cases it is desirable to pass new array elements by using constants or expressions. The TPVARRAY unit cannot directly support this because an array's data type is not known at compile time. Fortunately, it is very easy for your application to define one or more such procedures to set the specific data types of your application. Here's an example:

```
procedure SetVal(A : TpArray; R, C : Word; V : MyDataType);
begin
  SetA(A, R, C, V);
end;
```

SetVal can then be called with constants or expressions in the position for V.

Note also that TPVARRAY provides no direct means to change a single field within a record type array element. See the entry for RetA in the TPRARRAY section for a way to work around this limitation.

Examples

```
SetA(A, 0, 0, W);
```

Sets the very first element in array A to the variable W

```
SetA(A, TotalRows-1, TotalCols-1, W);
```

Sets the very last element from array A

See Also

ErrorA RetA

StoreA

Declaration

```
procedure StoreA(A : TpArray);
```

Purpose

Write array to disk and close file.

Comments

For virtual arrays, StoreA performs the same operation as FlushA, and additionally closes the array file. Since most of a virtual array is already kept in a file, no additional filename needs to be specified here.

The final file size for the disk file can be computed as shown in the MakeA section.

StoreA will return an error if it is unable to write to or close the array file.

No further array operations can be performed with a stored array unless it is reloaded into memory. As a result, each call to StoreA should normally be followed with a call to DisposeA to deallocate the array's now useless heap space.

Example

```
StoreA(A)
```

Completely updates the disk image of array A and closes the disk file.

See Also

DisposeA

MakeA

11. BCD Arithmetic

A. Overview

There are times when exceptional precision is required for real number calculations. Obvious examples are financial calculations, where a billion dollar sum must be accurate to the penny, or an amortization table that must retain at least 10 accurate digits for 30 years.

Turbo Pascal's 6-byte real data type does not provide sufficient accuracy for these situations. Two problems arise: first, there just isn't space in 6 bytes to retain sufficient precision; and second, because storage is in binary format, roundoff errors occur when converting numbers to and from decimal format.

Turbo Pascal version 3.0 provided a special version of the compiler just to solve this problem. This compiler defined a real type with 10 byte BCD (Binary Coded Decimal) format, providing 18 digits of accuracy and a range of 1E-63 to 1E+63.

Turbo Pascal version 4.0 and higher no longer support BCD arithmetic. Although 8087 data types supporting up to 19 significant digits are available, these reals still use a binary format, leading to roundoff errors.

The Turbo Professional TPBCD unit offers an extended set of BCD arithmetic routines. A BCD real is represented as an array[1..10] of byte, with the same internal format as Turbo Pascal 3.0's BCD reals. Besides a full complement of arithmetic, comparison and format conversion routines, TPBCD provides the previously missing group of transcendental functions: Sqrt, Exp, Ln, Sin, Cos, and ArcTan. In addition, TPBCD includes a Form function for easy formatting of numeric results.

The TPBCD routines are coded in assembly language for best speed and minimum code size. If no transcendental routines are used, TPBCD will add about 3K to a program's code size. If any transcendental routine is called, an additional 3K of code will be pulled in by the linker.

The non-transcendental functions retain full 18-digit accuracy for all operations. The transcendental operations retain at least 17 digits of accuracy, and are typically within one digit in the 18th place.

Because BCD calculations are not infix operations, and because Pascal functions cannot return structured types (such as arrays), the process of computing with BCD reals is now a bit more cumbersome. All TPBCD routines that compute BCD real

results return them as VAR parameters. See the following section for detailed calling syntax.

Runtime errors in TPBCD are handled in a straightforward fashion. Whenever any numeric error occurs, whether it be divide by zero, floating point overflow, or an illegal operation such as the square root of a negative number, TPBCD generates a divide by zero interrupt (int 0). By default, the Turbo Pascal system error handler takes this interrupt, reports a "Divide by zero," and shuts down the program. The error address reported by Turbo Pascal will be somewhere within the TPBCD unit, making it difficult to determine the true source of the problem. Optionally, you may use the TPERRHAN unit, in which case all TPBCD numeric errors are recoverable. By polling ExitCode after error recovery, you can make a more accurate determination of the true error location.

For an example of how to use the routines in TPBCD, see the demonstration program PCALC.PAS. PCALC is a memory-resident programmer's calculator that, when used in conjunction with SMACS, can paste its results back into your application program. It can even translate real numbers into a typed constant array format that is ideal for specifying BCD constants in your source code. PCALC also demonstrates a useful technique for trapping arithmetic errors reported by the routines in TPBCD.

Special thanks to Philip R. Burns for his aid with the algorithms used in the transcendental functions.

B. TPBCD: High-precision Reals

This section describes all of the types, variables, constants, procedures and functions found in the interface section of the TPBCD unit.

Constants

```
BCDsize = 10;
```

Specifies the number of bytes used to store a BCD real. The value 10 is consistent with the BCD reals used by Turbo Pascal 3.0. By changing this constant and an equivalent constant in TPBCDLOW.ASM, BCD reals of higher accuracy can be obtained for all but the transcendental operations. (Note that you need Borland's Turbo Assembler or MicroSoft's Macro Assembler in order to reassemble TPBCDLOW.ASM.) Ten byte-BCD reals offer 18 digits of accuracy, and two additional digits of accuracy would be gained for each extra byte used. Transcendental operations do not necessarily gain in accuracy because the algorithms to compute the functions are designed to provide only 18 digits of precision. More accurate algorithms would require more time and space than the existing ones.

```
CommaForPeriod : Boolean = False;
```

If True, the Form function will return '.' for ',' and vice versa. This makes it a bit easier to write programs that can be used both in the U.S. and in countries that use commas for decimal points.

```
MoneySign : Char = '$';
```

This character is used when displaying values in dollars-and-cents format. If desired, you could change this constant to #156, for example, allowing Form to display amounts in terms of British pounds rather than dollars.

```
PiBcd : BCD = ($40, $24, $93, $97, $58, $53, $26, $59, $41, $31);
```

Specifies Pi to 18 digits in BCD format.

```
ZeroBcd : BCD = ($0, $0, $0, $0, $0, $0, $0, $0, $0, $0);
```

0 in BCD format.

Types

```
BCD = array[1..BCDsize] of Byte;
```

The internal format of a BCD real. See TPBCD.ASM for a more detailed description of the BCD format.

AbsBCD / AddBCD

Declaration

```
procedure AbsBCD(B1 : BCD; var B2 : BCD);
```

Purpose

Returns absolute value of B1 in B2.

Comments

This is a very fast operation that requires clearing only one bit in the input real, and transferring ten bytes to the output.

Example

```
{Assume B1 holds -1.2345}
AbsBCD(B1, B2);
```

B2 returns 1.2345

Declaration

```
procedure AddBCD(B1, B2 : BCD; var B3 : BCD);
```

Purpose

Add B1 to B2 and put result in B3.

Comments

AddBCD creates a temporary result with 19 digits of accuracy, then rounds this back to 18 digits if required.

If the result exceeds 1.0E64, an overflow error occurs.

Example

```
{Assume that B1 holds 0.999999999999999999}
AddBCD(B1, B1, B2);
```

B2 then contains 2.000000000000000000 because the intermediate result contains 19 digits and is rounded up.

Declaration

```
procedure ArcTanBCD(B1 : BCD; var B2 : BCD);
```

Purpose

Returns arc tangent of B1 in B2.

Comments

Like the equivalent operation with normal reals, ArcTanBCD returns the angle, in radians, whose tangent is B1. ArcTanBCD always returns values between -Pi/2 and +Pi/2. Any value within the range of BCD reals is acceptable as an input parameter.

Example

```
{Assume that B1 holds 1.0}
ArcTanBCD(B1, B2);
{Assume that B3 holds 4.0}
MultBCD(B2, B3, B4);
```

Then B4 returns 3.14159265358979323, which equals Pi to within one digit in the last place.

Declaration

```
function BCDtoReal(B : BCD) : Real;
```

Purpose

Convert a BCD to a real.

Comments

This routine converts a BCD to a 6-byte non-8087 real. We assume that you wouldn't be using the TPBCD unit if you were already using 8087 reals.

The routine works by converting a BCD to the equivalent full-precision ASCII string (a fast operation), and then calling the SYSTEM unit VAL procedure to convert the string to a real. The 18 digits of BCD accuracy are rounded to the 11 digits of Turbo accuracy.

See Also

LongIntToBCD RealToBCD

CosBCD / DivBCD

Declaration

```
procedure CosBCD(B1 : BCD; var B2 : BCD);
```

Purpose

Returns cosine of B1 in B2.

Comments

Like Turbo's normal Cos function, CosBCD takes an angle expressed in radians and returns the cosine in parameter B2. The largest acceptable input value is 3.14E9 radians, and the smallest is -3.14E9. The output value is always in the range -1.0 to +1.0.

CosBCD is accurate within three digits in the 18th place for all cases where we've found a guaranteed precise reference. It also matches the results of an 80287 numeric coprocessor within three digits in the 18th place for a broad range of input values.

Example

```
CosBcd(PiBcd,B1);
```

Returns in B1 the value -0.999999999999999997 (should be -1.0)

See Also

SinBcd

Declaration

```
procedure DivBCD(B1, B2 : BCD; var B3 : BCD);
```

Purpose

Divide B1 by B2 and put result in B3.

Comments

The division algorithm used is analogous to the "subtract and shift" technique taught in grade school. It is precise for all 18 digits.

If the result exceeds 1.0E64, an overflow error occurs. If the result is less than 1.0E-63, zero is returned.

See Also

MultBCD

Declaration

```
function EqualBCD(B1, B2 : BCD) : Boolean;
```

Purpose

Returns True if B1 = B2.

Comments

EqualBcd compares the two operands for exact equality. The smallest difference will cause it to return False.

To compare for approximate equality, you should define a function like the following:

```
function ApproxEqual(B1, B2, Epsilon : BCD) : Boolean;
  {-Return true if Abs(B1-B2) < Epsilon}
var
  Difference : BCD;
begin
  SubBCD(B1, B2, Difference);
  AbsBCD(Difference, Difference);
  ApproxEqual := LessBCD(Difference, Epsilon);
end;
```

To see if two items matched within one tenth of a cent, you would make a call like:

```
RealToBcd(0.001, OneTenthCent);
if ApproxEqual(B1, B2, OneTenthCent) then
  ...;
```

See Also

NotEqualBCD

ExpBCD / Form

Declaration

```
procedure ExpBCD(B1 : BCD; var B2 : BCD);
```

Purpose

Returns the exponential of B1 in B2.

Comments

The largest acceptable value of B1 is 147.36. Values larger than this will cause an overflow error.

The smallest value of B1 for which the result retains any resolution is -145.06. ExpBCD returns 0.0 for values smaller than this.

ExpBCD is accurate within 1 digit in the 18th place for all cases where we've found a guaranteed precise reference. It matches the results of an 80287 coprocessor within 3 digits in the 18th place across a broad range of inputs.

Example

```
{Assume B1 contains 1.0}
ExpBCD(B1, B2);
```

Returns in B2 the value 2.71828182845904524 (e accurate to 18 digits).

See Also

LnBcd

Declaration

```
function Form(Mask : string; B : BCD) : string;
```

Purpose

Returns a formatted string with digits from B merged into the Mask.

Comments

Form provides the most important features of the Turbo 3.0 BCD Form function. This function offers advanced formatting capabilities for real numbers. It is the only TPBCD function written in Pascal--if you wish, it should be easy to modify or add features.

See the description of the Form function in the TPSTRING section for complete information.

See Also

Form (TPSTRING)
StrBCD
StrExpBCD

Declaration

```
procedure FracBCD(B1 : BCD; var B2 : BCD);
```

Purpose

Returns the fractional part of B1 in B2.

Comments

FracBCD's result is the same as computing FracBCD := B1-IntBCD(B1).

Examples

```
{Assume B1 contains 1.46}
FracBCD(B1, B2);
```

Returns 0.46 in B2

```
{Assume B1 contains -1.46}
FracBCD(B1, B2);
```

Returns -0.46 in B2

Declaration

```
function GreaterBCD(B1, B2 : BCD) : Boolean;
```

Purpose

Returns True if B1 > B2.

Comments

All of the BCD comparison functions share a common low level routine which does the actual comparison and returns the 8086 flags set for the result. The top level function then interrogates the flags to return an appropriate boolean value.

The comparison works with all 18 digits. Thus 1.00000000000000001 is greater than 1.00000000000000000.

GreaterEqualBCD / IntBCD

Declaration

```
function GreaterEqualBCD(B1, B2 : BCD) : Boolean;
```

Purpose

Returns True if B1 >= B2.

Comments

All of the BCD comparison functions share a common low level routine which does the actual comparison and returns the 8086 flags set for the result. The top level function then interrogates the flags to return an appropriate boolean value.

The comparison works with all 18 digits.

Declaration

```
procedure IntBCD(B1 : BCD; var B2 : BCD);
```

Purpose

Returns the integer part of B1 in B2.

Comments

If B1 is greater than or equal to zero, then IntBcd returns the greatest integer less than or equal to B1. If B1 is less than zero, then IntBCD returns the smallest integer greater than or equal to B1.

Note that the result is a BCD real with no fractional part, not an actual integer.

Examples

```
{Assume B1 contains 1.46}
IntBCD(B1, B2);
```

Returns 1.0 in B2

```
{Assume B1 contains -1.46}
IntBCD(B1, B2);
```

Returns -1.0 in B2

Declaration

```
function LessBCD(B1, B2 : BCD) : Boolean;
```

Purpose

Returns True if B1 < B2.

Comments

All of the BCD comparison functions share a common low level routine which does the actual comparison and returns the 8086 flags set for the result. The top level function then interrogates the flags to return an appropriate boolean value.

The comparison works with all 18 digits.

Declaration

```
function LessEqualBCD(B1, B2 : BCD) : Boolean;
```

Purpose

Returns True if B1 <= B2.

Comments

All of the BCD comparison functions share a common low level routine which does the actual comparison and returns the 8086 flags set for the result. The top level function then interrogates the flags to return an appropriate boolean value.

The comparison works with all 18 digits.

LnBCD / LongintToBCD

Declaration

```
procedure LnBCD(B1 : BCD; var B2 : BCD);
```

Purpose

Returns the natural logarithm of B1 in B2.

Comments

The input to LnBcd must be a positive number or a numeric error will occur.

LnBCD is accurate within one digit in the 18th place for all cases where we've found a guaranteed precise reference. It matches the results of an 80287 numeric coprocessor within two digits in the 18th place for a broad range of inputs.

Examples

```
{Assume B1 contains 2.0}
LnBCD(B1, B2);
```

Returns 6.93147180559945308E-01 in B2 (last digit should be 9).

See Also

ExpBCD

Declaration

```
procedure LongintToBCD(L : LongInt; var B : BCD);
```

Purpose

Convert a LongInt to a BCD.

Comments

This procedure provides a convenient method to convert integer constants to BCD values. The BCD holds all 10 digits of the largest LongInt exactly.

Examples

```
LongintToBCD(0, B1);
```

Puts the value 0 in B1

```
LongintToBCD(MaxLongInt, B1)
```

Puts 2147483647 in B1.

See Also

RealToBCD

Declaration

```
procedure MultBCD(B1, B2 : BCD; var B3 : BCD);
```

Purpose

Multiply B1 by B2 and put result in B3.

Comments

The multiplication algorithm used is analogous to the "multiply digits and carry" technique taught in grade school. It is precise for all 18 digits.

If the result exceeds 1.0E64, an overflow error occurs. If the result is less than 1.0E-63, zero is returned.

See Also

DivBCD

Declaration

```
function NotEqualBCD(B1, B2 : BCD) : Boolean;
```

Purpose

Returns True if B1 <> B2.

Comments

NotEqualBcd returns True unless the inputs agree in all 18 digits. Use caution in applying such comparisons.

To allow for approximate equality, you should define a function like the following:

```
function NotApproxEqual(B1, B2, Epsilon : BCD) : Boolean;
  {-Return true if Abs(B1-B2) > Epsilon}
var
  Difference : BCD;
begin
  SubBCD(B1, B2, Difference);
  AbsBCD(Difference, Difference);
  ApproxEqual := GreaterBCD(Difference, Epsilon);
end;
```

See Also

EqualBCD

RealToBCD

Declaration

```
procedure RealToBCD(R : Real; var B : BCD);
```

Purpose

Convert a real to a BCD.

Comments

This procedure provides a convenient method for initializing BCD real constants. As long as the precision of the constant does not exceed that of a 6-byte real, the BCD will receive an exact constant value.

If you have constants that require more than 11 digits of accuracy, we suggest that you use the supplied demonstration program PCALC (together with SMACS) to generate typed constants in BCD array format. In this way, the constants can be accurate in all 18 digits. Alternatively, you may use the ValBCD procedure to convert a string representation of the constant to a BCD real.

RealToBCD works *only* to convert Turbo's 6-byte reals. It will *not* work with 8087 format reals. We assume that you wouldn't be using the TPBCD unit if you were already using 8087 reals.

Examples

```
RealToBCD(1.5, B1);
```

Initializes B1 to 1.5

```
RealToBCD(-1.78234976E-22, B1)
```

Initializes B1 to -1.78234976E-22

See Also

LongIntToBCD

Declaration

```
function RoundBCD(B1 : BCD) : LongInt;
```

Purpose

Returns the value of B1 rounded to the nearest long integer.

Comments

If B1 exceeds MaxLongInt (2,147,483,647), then RoundBCD triggers an overflow error. Otherwise it performs normal rounding to generate a LongInt result.

If B1 is greater than or equal zero, then RoundBCD returns TruncBCD(B1+0.5). Otherwise, it returns TruncBCD(B1-0.5).

Examples

```
{Assume B1 contains 1.46}
RoundBCD(B1);
```

Returns 1

```
{Assume B1 contains 1.50}
RoundBCD(B1)
```

Returns 2

```
{Assume B1 contains -1.46}
RoundBCD(B1);
```

Returns -1

```
{Assume B1 contains -1.50}
RoundBCD(B1);
```

Returns -2

See Also

TruncBCD

SinBCD

Declaration

```
procedure SinBCD(B1 : BCD; var B2 : BCD);
```

Purpose

Returns the sine of B1 in B2.

Comments

Like Turbo's normal Sin function, SinBCD takes an angle expressed in radians and returns the sine in parameter B2. The largest acceptable input value is 3.14E9 radians, and the smallest is -3.14E9.

SinBCD is accurate within one digit in the 18th place for all cases where we've found a guaranteed precise reference. It also matches the results of an 80287 numeric coprocessor within one digit in the 18th place for a broad range of inputs.

Example

```
RealToBcd(4.0, B1);
DivBcd(PiBCD, B1, B2); {Compute Pi/4}
SinBcd(B2, B4);
```

Returns Sin(Pi/4) in B4 = 7.07106781186547525E-01

See Also

CosBCD

Declaration

```
procedure SqrBCD(B1 : BCD; var B2 : BCD);
```

Purpose

Returns the square of B1 in B2.

Comments

This operation is equivalent to MultBCD(B1, B1, B2). It runs a bit faster and is slightly easier to call.

If the result exceeds 1.0E64, an overflow error occurs. If the result is less than 1.0E-63, zero is returned.

See Also

MultBCD

Declaration

```
procedure SqrtBCD(B1 : BCD; var B2 : BCD);
```

Purpose

Returns the square root of B1 in B2.

Comments

The input must be non-negative or a numeric error occurs.

The algorithm used is Heron's approximation. It is accurate for all 18 digits and requires 4 iterations.

Example

```
{Assume B1 holds 2.0}
SqrtBCD(B1, B2);
```

Returns 1.41421356237309505, accurate to 18 digits.

StrBCD

Declaration

```
function StrBCD(B : BCD; Width, Places : Byte) : string;
```

Purpose

Return a BCD as a string.

Comments

This routine is functionally equivalent to the Turbo Pascal syntax

```
Str(R:Width:Places, S)
```

StrBCD returns a string in floating point format (no exponent value is included). Note that StrBCD is a function rather than a procedure like Str.

If you wish to obtain a BCD real in exponential format, use the function StrExpBCD. The Form function provides yet another way to format BCD reals for output.

The maximum useful number of decimal Places to specify is 18, although StrBCD will return more (filled with zeros) if you ask for them. If the actual width of the string exceeds the requested Width, StrBcd will use the actual width. If the actual width is less, the answer is right justified in a string of the requested width.

Examples

```
WriteLn(StrBCD(B1, 10, 2));
```

Writes the BCD real B1 to the screen the same way that

```
WriteLn(R1:10:2);
```

would write it for a normal real.

```
StrBCD(B1, 0, 0);
```

Returns the real B1 rounded to the nearest integer in a string of minimum width.

See Also

Form

StrExpBCD

Declaration

```
function StrExpBCD(B : BCD; Width : Byte) : string;
```

Purpose

Return B as a string in exponential format.

Comments

This routine formats a BCD real as an exponential format string. If Width is 0, the result is a 24-character string of the form -n.nnnnnnnnnnnnnnnnE+ee; the first character is a blank for non-negative numbers. If Width is not 0, the string returned will be Width characters long. If Width is less than 24, the excess digits will be rounded; if it is greater than 24, the value will be right justified in the string, which will be padded at the front with blanks. If Width is greater than 0 and less than 8, a Width of 8 will be used.

Example

```
{Suppose B1 has 123456.9}
StrExpBCD(B1, 0)
```

Returns ' 1.23456900000000000E+05'

```
StrExpBCD(B1, 9)
```

Returns ' 1.23E+05'

See Also

Form StrBCD

Declaration

```
procedure SubBCD(B1, B2 : BCD; var B3 : BCD);
```

Purpose

Subtract B2 from B1 and put result in B3.

Comments

SubBCD creates a temporary result with 19 digits of accuracy, then rounds this back to 18 digits if required.

If the result exceeds 1.0E64, an overflow error occurs. If the result is less than 1.0E-63, zero is returned.

TruncBCD

Declaration

```
function TruncBCD(B1 : BCD) : LongInt;
```

Purpose

Returns the greatest long integer less than or equal to B1.

Comments

If B1 is greater than or equal to zero, TruncBCD returns the greatest integer less than or equal to B1. Otherwise, TruncBCD returns the smallest integer greater than or equal to B1.

If B1 exceeds MaxLongInt (2,147,483,647), an overflow error occurs.

Examples

```
{Assume B1 contains 1.46}
TruncBCD(B1);
```

Returns 1

```
{Assume B1 contains 1.999999}
TruncBCD(B1)
```

Returns 1

```
{Assume B1 contains -1.999999}
TruncBCD(B1);
```

Returns -1

See Also

RoundBCD

Declaration

```
procedure ValBCD(S : string; var B : BCD; var Code : Word);
```

Purpose

Convert a string to a BCD.

Comments

This procedure provides for BCD reals the equivalent of Turbo's Val routine. S is a string in any valid floating point or exponential format. Leading blanks are acceptable, but trailing blanks are not.

If the conversion is successful, Code will contain zero, and B will hold the converted real. Otherwise, Code will contain a non-zero value that points to the input string location causing the error, and B will contain an indeterminate value. If the string specifies a real too large to fit into BCD format, Code will point beyond the end of the string.

Examples

```
ValBCD('  1.0E22  ', B1, Code);
```

Returns Code=9 because trailing blanks are not allowed.

```
ValBCD('-1.0E99', B1, Code);
```

Returns Code=8 because the specified value exceeds the range of BCD reals.

```
ValBCD('0.57721566490153286061', B1, Code);
```

Returns Code=0 and the value of Euler's constant loaded into B1. Note that excess digits are truncated to fit into the 18 digit mantissa of BCD reals.

See Also

Form
StrBCD
StrExpBCD

12. Miscellaneous

A. Overview

This chapter is devoted to smaller units that don't fit readily into any of the general groups already defined. Currently there are only two such units. The TPERRHAN unit provides a facility that allows you to recover from almost all runtime errors. The TPINLINE unit contains a collection of "inline procedures." (We call them *inline macros*.) Besides being useful in themselves, these short routines demonstrate some useful techniques that may help you write your own inline macros.

B. TPERRHAN: Runtime Error Recovery

Turbo Pascal offers a significant measure of flexibility in handling runtime errors. The programmer can disable range checking, stack overflow checking, and I/O result checking. Heap allocation errors can be diverted by using the new HeapError facility. All runtime errors can be trapped to a user-defined exit procedure.

All of this leaves one question unanswered: how can a program recover from a runtime error and continue processing? Turbo Professional provides the answer with the TPERRHAN unit.

TPERRHAN works by tapping into the Turbo Pascal runtime error handler. Control is diverted from the normal SYSTEM unit to a routine defined by TPERRHAN. The TPERRHAN routine classifies errors as recoverable or fatal, computes where control should return upon recovery, optionally calls a user-defined function, and finally returns to the program or to the SYSTEM error handler as required.

Some runtime errors are hopelessly bad--TPERRHAN recognizes these and passes control back to the SYSTEM error handler, which shuts down the program in an orderly fashion after calling any user-defined exit procedures. We classify several Turbo runtime errors as fatal: error 202 (Stack overflow), error 203 (Heap overflow), errors 208 and 209 (overlay errors), any error when the 80x87 numeric coprocessor (or Turbo Pascal's numeric emulator) is involved, and error 216 (general protection fault) in protected mode. All other Turbo Pascal runtime errors, including I/O errors, are considered recoverable.

Stack overflow problems can be solved by increasing the program's stack segment size via the {$M} compiler directive or by reorganizing the program's use of local variables. You can manage heap overflow errors in a recoverable way by using the Turbo Pascal HeapError facility.

You can manage numeric coprocessor errors in a recoverable way by programming the 80x87's status register to return error codes rather than generating exception interrupts. See the TP8087 unit for details.

Note that I/O errors (that is, runtime errors whose number is less than 200) will leave IoResult set to a non-zero value. IoResult must be cleared (by reading it) before additional I/O operations can be successfully executed. Because you can turn off Turbo's I/O checking with the {$I-} compiler directive, it is unlikely that you will need to use TPERRHAN for handling I/O errors. In some situations, however, the option of a centralized error handler may be desirable.

The following program demonstrates use of the TPERRHAN unit.

```
{$R+}
program TestErrHan;
uses Dos, TpErrHan, TpString;
var
  I, J, K : Integer;
  V : 1..10;

  {$F+} {Must use the FAR model}
  function RecoverError : Boolean;
    {-Invoked when an error occurs. Return true to enable recovery}
  var
    I : Word;
  begin
    {Clear ioresult}
    I := IoResult;
    {Reinit std output}
    Assign(Output, '');
    Rewrite(Output);
    WriteLn('error handler sees error ', ExitCode:3, ' at ',
            HexW(Seg(ErrorAddr^)), ':', HexW(Ofs(ErrorAddr^)));
    RecoverError := True;
  end;
  {$F-}

begin
  {Enable error recovery, and install user routine}
  ErrorRecovery(True, @RecoverError);

  I := 0;   J := 1;   K := J div I;
  if ExitCode <> 0 then WriteLn('Divide by zero error occurred');

  I := 20;  V := I;
  if ExitCode <> 0 then WriteLn('Range error occurred');
end.
```

Variables

```
CanRecover : Boolean;
```

Because TPERRHAN must patch into Turbo's runtime system, its initialization code checks carefully to find the right patch point. If all goes well, TPERRHAN sets CanRecover to True. Presently we know of only two reasons why CanRecover might be False: either the Turbo SYSTEM library has changed radically as regards error handling, or the application program has taken control of int $0 (divide by zero) or int $23 (control break) prior to the time when TPERRHAN's initialization code is executed. The first problem necessitates changes to the TPERRHAN initialization code. The second can be solved by reordering the application's USES statement to place TPERRHAN before any user-defined units.

ErrorRecovery

Declaration

```
procedure ErrorRecovery(On : Boolean; UserRoutine : Pointer);
```

Purpose

Turn error recovery on or off, and attach to a user error routine.

Comments

Call this procedure to activate or deactivate runtime error recovery. If the On parameter is True, error recovery is activated; otherwise it is deactivated and Turbo Pascal's normal error handling system is used.

TPERRHAN can recover from most but not all runtime errors. See the introduction to TPERRHAN for more details.

UserRoutine is a pointer to a user-defined function which can further customize the error handling behavior of an application. If you don't want to define such a user function, simply pass the value nil in this position.

UserRoutine must point to a function declared as follows:

```
{$F+}
function RecoverError : Boolean;
  {-Invoked when an error occurs. Return true to enable recovery}
begin
  ...
end;
{$F-}
```

Note that the routine *must* be a global procedure compiled under the FAR model. Not preceding the function with the {$F+} directive will lead to a program crash.

The UserRoutine function will be called whenever a recoverable runtime error occurs. At that time the ExitCode global byte will contain the runtime error number, and the ErrorAddr global pointer will contain the normalized address where the runtime error occurred. UserRoutine can evaluate these variables and take any action required. Before exiting, UserRoutine should set its result True if recovery is desired, and False if the program should be terminated.

If a runtime error is fundamentally not recoverable, or if the UserRoutine returns False, then TPERRHAN passes control to the normal system error handler. This handler will operate as usual by calling any user-defined exit procedures, closing all files, restoring interrupt vectors, and halting.

If a runtime error is recoverable and the UserRoutine, if any, returns True, then TPERRHAN's routines will return control to the machine instruction immediately following the one that caused the error. The application program is responsible for regularly polling the ExitCode byte to determine whether an error has occurred. ExitCode will contain the normal values associated with Turbo runtime errors: zero for success, non-zero for failure. See the Turbo Pascal manual for a detailed listing of the error codes.

Whenever an application uses Turbo Professional's error recovery system, TPERRHAN's initialization code automatically calls ErrorRecovery to activate error recovery. By default, TPERRHAN installs no UserRoutine.

Example

```
    {$F+}
    function RecoverError : Boolean;
    var I : Word;
    begin
      I := IoResult;            {Clear IoResult}
      RecoverError := True;     {Recover from all errors}
    end;
    {$F-}
  begin
    ErrorRecovery(True, @RecoverError);
    ...
  end.
```

This is the simplest useful UserRoutine. The routine clears IoResult to allow further I/O operations to proceed, and returns a True result to recover from all forms of runtime errors.

See Also

Error8087 (TP8087)

Exceptions8087 (TP8087)

C. TPINLINE: Assorted Inline Macros

Among the most powerful of the features introduced in Turbo Pascal 4.0 were its "inline procedures." Unlike normal procedures and functions, which reside at a fixed location in memory, the entire contents of an inline procedure are inserted directly into the object code produced by the compiler each time it is referenced. Although inline procedures can take parameters and return function results, just like regular procedures and functions, the compiler treats them more like symbolic macros, such as those available in C. For this reason, we prefer to call them *inline macros*.

Among other things in TPINLINE, you will find inline macros that allow you to:

- Set a "jump marker" and later return to the location where the marker was set, from virtually any place in your program
- Call any procedure or function that takes no parameters, allowing you to easily construct jump tables
- Manipulate the individual words that make up a long integer, just as Hi, Lo, and Swap allow you to work with the individual bytes that make up a word
- Normalize pointers
- Set, clear, and test bit flags
- Exchange the values in two variables without using a temporary variable
- Initialize an array of any type without using a FOR loop

Types

```
JumpRecord =
  record
    SpReg, BpReg : Word;
    JmpPt : Pointer;
  end;
```

Used by SetJump and LongJump to store data used to perform non-local goto's.

Declaration

```
procedure CallOldIsr(OldIsr : Pointer);
```

Purpose

Call previous ISR from an interrupt procedure.

Comments

Simulates an interrupt by pushing the CPU flags and calling FAR to the address passed as OldIsr. Like EmulateInt (TPINT), this macro thus allows you to call the ISR that serviced a particular interrupt vector before you grabbed it. Unlike EmulateInt, however, it is intended primarily for use in ISR's written in inline code, rather than Pascal. Note that the BX register is destroyed by this macro.

Example

```
GetIntVec($09, SaveInt09);
...
procedure MyIsr(BP : Word); interrupt;
begin
  CallOldIsr(SaveInt09);
  {-- something --}
end;
```

Calls the previous interrupt $09 handler.

See Also

EmulateInt (TPINT) JumpToOldIsr

Declaration

```
procedure ClearFlag(var Flags : Word; FlagMask : Word);
```

Purpose

Clear the bit(s) specified by FlagMask in Flags.

Comments

This macro clears all bits in the Flags word that are set in FlagMask. Note that the Flags parameter *must* be a word variable.

Example

```
ClearFlag(BitFlags, $02);
```

Clears bit 1 in BitFlags ($02 = 0000000000000010 in binary).

See Also

FlagIsSet SetFlag

ExchangeBytes / ExchangeStructs

Declaration

```
procedure ExchangeBytes(var I, J : Byte);
```

Purpose

Exchange bytes I and J.

Comments

This macro allows you to exchange the values in I and J without using a temporary variable. It is especially useful in sorts.

Example

```
if MyArray[I] > MyArray[I+1] then
  ExchangeBytes(MyArray[I], MyArray[I+1]);
```

Exchange MyArray[I] and MyArray[I+1] if MyArray[I] is greater.

See Also

ExchangeWords

Declaration

```
procedure ExchangeStructs(var I, J; Size : Word);
```

Purpose

Exchange structures I and J. Useful in sorts.

Comments

This macro allows you to exchange the values in I and J without using a temporary variable. I and J may be variables of any type, but they must be the same Size. It is especially useful in sorts.

Examples

```
if MyRec1.Field > MyRec2.Field then
  ExchangeStructs(MyRec1, MyRec2, SizeOf(MyRec1));
```

Exchange MyRec1 and MyRec2 if MyRec1.Field is greater than MyRec2.Field.

```
if MyString1 > MyString2 then begin
  I := MaxWord(Length(MyString1), Length(MyString2));
  ExchangeStructs(MyString1, MyString2, I+1);
end;
```

Exchange MyString1 and MyString2 if MyString1 is greater. Notice that the number of bytes moved is limited to the length of the longer string plus 1 (to account for the length bytes of the two strings).

See Also

ExchangeWords MaxWord

Declaration

```
procedure ExchangeWords(var I, J : Word);
```

Purpose

Exchange words I and J. Useful in sorts.

Comments

This macro allows you to exchange the values in I and J without using a temporary variable. It is especially useful in sorts.

Example

```
if MyArray[I] > MyArray[I+1] then
  ExchangeWords(MyArray[I], MyArray[I+1]);
```

Exchange MyArray[I] and MyArray[I+1] if MyArray[I] is greater.

See Also

ExchangeBytes

ExchangeStructs

Declaration

```
procedure FarCall(ProcAddr : Pointer);
```

Purpose

ProcAddr is the address of a routine to be called FAR.

Comments

This macro lets you call any procedure in any code segment as long as 1) it takes no parameters, 2) it was compiled under the FAR model using {$F+}, and 3) it is not nested inside another procedure. FarCall might be used, for example, to implement a jump table. Obviously, this macro is most useful in Turbo version 4.0, which does not implement procedure variables.

Examples

```
ProcPtr := @MyProc;
...
FarCall(ProcPtr^);
```

Incorrect! The '^' is not needed.

```
FarCall(ProcPtr);
```

Correct. This would make a FAR call to MyProc.

```
var MyJumpTable : array[1..10] of Pointer;
...
FarCall(MyJumpTable[Index]);
```

Make a FAR call to the procedure whose address is in MyJumpTable[Index].

See Also

NearCall

FillStruct / FillWord

Declaration

```
procedure FillStruct(var Dest; Count : Word; var Filler; FillerSize : Word);
```

Purpose

Fill memory starting at Dest with Count instances of Filler.

Comments

This macro is similar to the standard FillChar procedure, except that it fills memory with a value of FillerSize bytes rather than a byte-sized value. Dest is the location in memory to start filling at. Filler is the value to fill with. Its size is given by FillerSize. Count is the number of instances of Filler. FillStruct is especially useful for initializing arrays. Note that no range checking is done to prevent segment overruns.

Example

```
var   MyArray : array[1..1000] of Real;
const R : Real = 1.2345;
...
FillStruct(MyArray, 1000, R, SizeOf(R));
```

Initialize all 1000 elements of MyArray to 1.2345.

Declaration

```
procedure FillWord(var Dest; Count, Filler : Word);
```

Purpose

Fill memory starting at Dest with Count instances of Filler.

Comments

This macro is similar to the standard FillChar procedure, except that it fills memory with a word-sized value rather than a byte-sized value. Dest is the location in memory to start filling at. Filler is the word value to fill with. And Count is the number of words to fill. FillWord is especially useful for initializing arrays. Note that no range checking is done to prevent segment overruns.

Example

```
var
  MyArray1 : array[1..100] of Word;
  MyArray2 : array[1..100] of Integer;
  I : Integer;
...
FillWord(MyArray1, 100, 1);
I := 1;
FillWord(MyArray2, 100, Word(I));
```

Initialize all 100 elements of MyArray1 and MyArray2 to 1.

Declaration

```
function FlagIsSet(Flags, FlagMask : Word) : Boolean;
```

Purpose

Returns True if the bit specified by FlagMask is set in Flags.

Comments

This macro tests to see if any of the bits set in FlagMask are also set in Flags. In most cases, FlagMask should not have more than one bit set. Although FlagIsSet can tell you if one of them is set, or none of them, it cannot tell you if all of them are set.

Examples

```
FlagIsSet(BitFlags,$03);
```

Probably incorrect. This will *not* tell you if bits 0 and 1 are both set ($03 = 0000000000000011 in binary). It will only tell you if one of them is set.

```
FlagIsSet(BitFlags,$02);
```

Correct. This would return True if bit 1 in BitFlags is set ($02 = 0000000000000010 in binary).

See Also

ClearFlag SetFlag

Declaration

```
function HiWord(L : LongInt) : Word;
```

Purpose

Return high-order word of L.

Comments

This macro returns the high-order word of a long integer. HiWord is found in the Macintosh version of Turbo Pascal, but not in the IBM version.

Example

```
const MyLongInt = $FFFF0000;
...
MyWord := HiWord(MyLongInt);
```

This would assign $FFFF to MyWord.

See Also

LoWord SwapWord

JumpToOldIsr

Declaration

```
procedure JumpToOldIsr(OldIsr : Pointer);
```

Purpose

Jump to previous ISR from an interrupt procedure.

Comments

Like ChainInt (TPINT), this macro allows you to restore the CPU registers pushed onto the stack at the beginning of an interrupt procedure, then pass on the interrupt to the ISR that serviced a particular vector before you grabbed it. Unlike ChainInt, however, it is intended primarily for use in ISR's written in inline code, rather than Pascal.

Examples

```
GetIntVec($09, SaveInt09);
...
procedure MyIsr(BP : Word); interrupt;
begin
  {-- something --}
  JumpToOldIsr(SaveInt09^);
end;
```

Incorrect! The '^' is not needed.

```
JumpToOldIsr(SaveInt09);
```

Correct. Jumps to the previous interrupt $09 handler.

See Also

CallOldIsr

ChainInt (TPINT)

Declaration

```
procedure LongJump(var JumpDest : JumpRecord);
```

Purpose

Restore SP, BP, and jump to JumpDest.JmpPt.

Comments

This macro uses the data stored in the JumpRecord by SetJump to perform a non-local goto. It restores the stack frame to the condition it was in when SetJump was called, then jumps to the instruction immediately after the call. Very important: LongJump cannot be used to jump from a higher stack frame into a lower one. Consider the following case, for example:

```
program MyProg;
  procedure MyProc;
  begin
    {Point B}
  end;
begin
  {Point A}
end.
```

It would be perfectly OK to call SetJump at Point A and later call LongJump from Point B to return to Point A. You could not, however, do the reverse (jump to B from A). Although LongJump uses a FAR jump instruction, the issue of FAR model {$F+} vs. NEAR model {$F-} does not arise here, as it does with call instructions.

Example

```
var MyJumpRecord : JumpRecord;
...
SetJump(MyJumpRecord);
WriteLn('This is JumpDest.JmpPt');
{-- something --}
LongJump(MyJumpRecord);
```

The call to LongJump would restart execution at the location of the WriteLn statement.

See Also

SetJump

LoWord / MaxWord

Declaration

```
function LoWord(L : LongInt) : Word;
```

Purpose

Return low-order word of L.

Comments

This macro returns the low-order word of a long integer. LoWord is found in the Macintosh version of Turbo Pascal, but not in the IBM version.

Example

```
const MyLongInt = $FFFF0000;
...
MyWord := LoWord(MyLongInt);
```

This would assign $0000 to MyWord.

See Also

HiWord

SwapWord

Declaration

```
function MaxWord(A, B : Word) : Word;
```

Purpose

Returns the greater of A and B.

Comments

This macro does an unsigned comparison of A and B and returns the greater of the two numbers.

Example

```
I := MaxWord(12, 40000);
```

I would be set to 40000.

See Also

MinWord

Declaration

```
function MinWord(A, B : Word) : Word;
```

Purpose

Returns the lesser of A and B.

Comments

This macro does an unsigned comparison of A and B and returns the lesser of the two numbers.

Example

```
I := MinWord(12, 40000);
```

I would be set to 12.

See Also

MaxWord

Declaration

```
procedure NearCall(ProcOfs : Word);
```

Purpose

ProcOfs is the offset of a routine to be called near.

Comments

This macro lets you call any procedure in the current code segment as long as 1) it takes no parameters, 2) it was compiled using the NEAR model (selected with the {$F-} compiler directive, the default setting), and 3) it is not nested inside another procedure. It might be used, for example, to implement a jump table.

Example

```
var MyJumpTable : array[1..10] of Word;
...
NearCall(MyJumpTable[I]);
```

Makes a NEAR call to the procedure whose offset is stored in MyJumpTable[I].

See Also

FarCall

Normalized / SetFlag

Declaration

```
function Normalized(P : Pointer) : Pointer;
```

Purpose

Return P as a normalized pointer.

Comments

This macro "normalizes" a pointer value. That is, it adjusts the segment and offset such that the offset is in the range $0000-$000F.

Examples

```
P1 := Ptr($0000, $0410);
P2 := Normalized(P1);
```

This would point P2 to $0041:$0000.

```
P1 := Ptr($0040, $0113);
P2 := Normalized(P1);
```

This would point P2 to $0051:$0003.

Declaration

```
procedure SetFlag(var Flags : Word; FlagMask : Word);
```

Purpose

Set the bit(s) specified by FlagMask in Flags.

Comments

This macro sets all bits in the Flags word that are set in FlagMask. Note that the Flags parameter *must* be a word variable.

Examples

```
SetFlag(BitFlags, $02);
```

Sets bit 1 in BitFlags ($02 = 0000000000000010 in binary).

```
SetFlag(BitFlags, $07);
```

Sets bits 2, 1, and 0 in BitFlags ($07 = 0000000000000111 in binary).

See Also

ClearFlag FlagIsSet

Declaration

```
procedure SetJump(var JumpDest : JumpRecord);
```

Purpose

Save current SP, BP, and a jump destination.

Comments

This macro saves data about the current stack frame and the instruction pointer, data which can later be used by LongJump to restart execution at the instruction immediately following the call to SetJump. A typical use for it might be to make a call to SetJump just before the start of the main program loop. Any time thereafter that an error occurred, you could jump back to the start of the loop using LongJump.

Example

```
var MyJumpRecord : JumpRecord;
...
SetJump(MyJumpRecord);
WriteLn('This is JumpDest.JmpPt');
{-- something --}
LongJump(MyJumpRecord);
```

The call to SetJump would store the data needed to allow LongJump to later restart execution at the location of the WriteLn statement.

See Also

LongJump

SwapNibble / SwapWord

Declaration

```
function SwapNibble(B : Byte) : Byte;
```

Purpose

Swap the high and low nibbles of B.

Comments

This routine is similar to the built-in Swap routine, but instead of swapping the high and low bytes of a word-sized value, it swaps the high and low nibbles of a byte-sized value. (A "nibble" is four bits, or half a byte.)

Examples

```
SwapNibble($F0)
```

Returns $0F (00001111 in binary; $F0 = 11110000 in binary).

```
SwapNibble($81)
```

Returns $18 (00011000 in binary; $81 = 10000001 in binary).

Declaration

```
function SwapWord(L : LongInt) : LongInt;
```

Purpose

Swap low- and high-order words of L.

Comments

This macro swaps the low- and high-order words of a long integer. SwapWord is found in the Macintosh version of Turbo Pascal, but not in the IBM version.

Example

```
const MyLongInt = $FFFF0000;
...
L := SwapWord(MyLongInt);
```

This would assign $0000FFFF to L.

See Also

HiWord LoWord

13. Demonstration Programs

As you are about to discover, Turbo Professional includes an impressive assortment of demonstration programs. Except in a couple of cases where our enthusiasm got the better of us (WINWOW, for example), we have aimed not only to provide realistic demonstrations of how to use the routines in the libraries, but also to give you some genuinely useful programs in the process. It is our belief that "real" programs will provide you with better models than would a series of tiny toys that did nothing more than show how to call a few of the routines in one of the units. As the name on the manual cover indicates, this is a "professional" package, and we think our demo programs help to prove it.

A. SMACS: Macro Processor

SMACS is a memory resident keyboard macro processor much like Borland's SuperKey. To load SMACS, enter

SMACS [MacroFile]

at the DOS prompt, where the optional MacroFile parameter is the name of a file containing previously recorded macros. SMACS comes with a sample macro file, TURBO5.MAC, which you might want to try using in conjunction with the Turbo Pascal editor.

Once SMACS is loaded, you can pop it up at almost any time by pressing <Alt><~>. When you do, you'll see a menu that looks something like this:

```
SUPER MACS: Macro Processor 5.00
Edit Delete Read Write List Status Configure Unload cApture Print
```

The following commands are available from the main menu:

Edit
Edit an existing macro or define a new one. When you issue this command, you will be asked first to press the key associated with the macro, and then for a name (optional) for the macro. Inside the macro editor, you can scroll through existing keystrokes using the cursor keys. By default the macro editor is in insert mode, so once the cursor has been positioned, you can insert new keys simply by pressing them. The <Ins> key toggles insert mode on and off. <Del> and <BkSp> delete macro keys. If you need to insert one or more keys that has a special function inside the macro editor--<Esc>, <Enter>, <Ins>, <Del>, the cursor keys, etc.--press <ScrollLock> to activate Literal mode, then press the key(s). To return to Command mode, press <ScrollLock> again. You can finish an editing session either by pressing <Enter>, which will save your changes, or <Esc>, which will not. For more details about the macro editor, see the section on the TPMACED unit.

Delete
Delete a previously defined macro or delete all macros. You will first be asked if you want to delete all macros. If you respond by pressing <Y>, you will be asked to confirm your choice. If you press <N> or <Enter>, you will be asked to press the key for the macro to be deleted. You can the press the appropriate key, or <Ctrl@> to cancel the operation.

Read
Read a macro file into memory. If there are any macros currently defined, you will

be asked whether the current ones should be deleted before the new ones are read in. Note that you can also load a macro file from the DOS command line or from a batch file. For example,

```
cd \turbo
smacs turbo5.mac
turbo
```

would load the sample macros from TURBO5.MAC and delete any existing macros *without asking for confirmation.*

Write
Save the currently defined macros to a macro file. If the file already exists, you will be asked if it should be overwritten.

List
Displays a scrollable list of all currently defined macros. The <Up> and <Down> keys will scroll the list if there are more macros than can be displayed on the screen at once. <Home> and <End> take you to the beginning and end of the list, respectively. And <Esc> returns you to the main menu.

Status
Displays the number of macros that are currently defined and the amount of memory available for new ones.

Configure
Displays a menu that allows you to see and change the configuration of SMACS:

Main menu
Change the hotkey used to pop up SMACS's main menu.

Begin macro
Change the hotkey used to turn macro recording on.

End macro
Change the hotkey used to turn macro recording off.

Screen reader
Change the hotkey that activates the screen reader.

Macro memory
Change the amount of memory set aside for macros.

Playback delay
Specify the delay factor to use when playing back macros. If the delay factor is non-zero, SMACS will wait for the specified number of milliseconds before returning a particular key in the macro. Such delays are sometimes needed to prevent keystrokes from being lost. The default setting is 0; the maximum is 100.

Macro playback
Toggle macro processing on and off.

Remap keyboard
If this option is on, you can assign macros to keys like <F11> (on an enhanced keyboard) and <AltUp> that normally cannot be associated with macros. See the documentation for the TPENHKBD unit for a complete list of the new keys that this option makes available.

Save configuration
Save the current settings permanently. SMACS.EXE must be in 1) the current directory, 2) the drive and directory it was loaded from originally (if you're using DOS 3.0 or higher), or 3) a directory on the DOS PATH.

Use the <Up> and <Down> cursor keys to select an option, and press <Enter> to select it. The first four choices let you change one of SMACS's hotkeys. There are some limitations on the hotkeys that you can assign in this fashion, but you may want to experiment with these commands anyway. Except for the Macro memory setting, any changes made in the configuration menu will go into effect immediately. Assuming that you use the Save configuration command to save your changes, the new Macro memory setting will go into effect the next time that SMACS is loaded.

Unload
Allows you, in many cases, to unload SMACS from memory without rebooting the system. SMACS will not unload itself if any program loaded above it uses the same interrupt vectors, so generally you cannot unload it unless it is the last program in memory. You can also unload SMACS from the command line by entering

```
SMACS -U
```

at the DOS prompt.

cApture
Allows you to capture the entire screen beneath the SMACS menu and save it to a file.

Print
Submit a file to the DOS 3.x (or higher) concurrent print facility. If PRINT.COM is not resident, or you are using DOS 2.x, an error message will be displayed.

Besides the commands available through the main menu, SMACS has three other commands that are available through hotkeys (shown in the second column).

Begin macro **<Alt><=>**
Turn macro recording on. You will be prompted for the key to assign the macro to. If a macro has already been assigned to the key, you will be asked if the existing macro should be overwritten. Also, if you specify a key like <A> or <+> that is used for normal typing, you will be asked to confirm your choice of keys.

End macro **<Alt><->**
Turn macro recording off. You will be asked for a name to assign to the macro; if you don't wish to assign it a name, simply press <Enter>.

Screen reader **<Alt><Space>**
Activate the screen reader. The screen reader allows you to capture a portion of the current screen (a "block" of up to 1000 characters) and store it in a macro. Use the cursor keys to position the cursor, then press <Enter> to mark the start of the block of characters. Then move the cursor to the end of the block and press <Enter> a second time to select it. As you move the cursor, the screen will be "painted" in reverse video to indicate the current boundaries of the block. Once the block has been marked and selected, you will be asked for the key to assign the macro to. To cancel the operation before a block has been selected, press <Esc>.

Note that only 1 of the 4 commands available through hotkeys may be used at a time. You cannot, for example, activate the screen reader from inside the main menu.

B. PCALC: Programmer's Calculator

PCALC is a memory resident calculator designed especially for programmers. It can operate in any of 5 different modes--2 for floating point calculations, 3 for integer calculations--and can perform 11 different arithmetic and logical operations. Integer calculations are performed using long integers in the range -2,147,483,648 ($80000000) to 2,147,483,647 ($7FFFFFFF). Floating point calculations are performed using BCD (Binary Coded Decimal) reals with 18 digits of precision and a range of 1E-63 to 1E+63.

To activate PCALC, press <Ctrl><RightShift><C>. What you'll see at the top right corner of the screen is a display that looks something like this:

```
┌─ PCALC: Programmer's Calculator 5.00 ─┐
│                              45.67    │
│                            1.9E-12 *  │
│                 0.000000000086773 =   │
├─ Float ──────────────────── Saved ────┤
│                                 0     │
├─ Commands ────────────────────────────┤
│ Clr Entry Binary Dec Hex Float exP Key │
│ Save Insert And Mod Not Or Xor shL shR │
└────────────────────────────────────────┘
```

The top three lines, the calculation window, are used for entering numbers and displaying the results of calculations. The second of the two is always used for entering numbers, and the third is always used to display results. To perform a calculation, follow these steps:

1. Enter the first number.
2. Press the appropriate key for the arithmetic or logical operation (+, -, *, /, A, M, N, O, X, L, or R).
3. Enter the second number.
4. Press <Enter> or <=> to display the result.
   - or -
   Press one of the keys listed in step 2 to complete the calculation and use the result as the first value in a second calculation.

Once you have a result, you can either start a new calculation by entering a number or press one of the keys listed in step 2 to use the result as the first value in a second calculation. Note that it is not necessary to press <NumLock> before entering numbers on the numeric key pad. PCALC automatically turns NumLock on when it is popped up, and restores the previous state of NumLock on exit.

The bottom two lines of the display show a menu of available commands. Just above it is the current value of the saved value buffer. There are two such buffers, one for real values and one for integers.

The following commands are available in PCALC:

Basic calculator commands:

<C> Clear
Clear the entire calculation window.

<E> clear Entry
Clear the current entry (second line of the calculation window).

<S> Save entry
Save the current entry in the appropriate saved value buffer.

<I> Insert saved value
Make the number in the saved value buffer the current entry.

Mode commands:

<B> Binary
Switch to binary mode (integer).

<D> Decimal
Switch to decimal mode (integer).

<H> Hexadecimal
Switch to hexadecimal mode (integer).

<F> Floating point
Switch to regular floating point mode (real). Results are displayed without exponents whenever possible--e.g., '0.0057' rather than '5.7E-03'.

<P> exPonential
Switch to exponential floating point mode (real). Results are always displayed in exponential format--e.g., '5.7E-03' rather than '0.0057'.

Arithmetic operators:

<+> Add
Add Value1 and Value2.

<-> Subtract
Subtract Value2 from Value1.

<*> Multiply
Multiply Value1 by Value2.

</> Divide
Divide Value1 by Value2.

<M> Mod
Remainder after dividing Value1 by Value2 (integer modes only).

Logical operators (integer modes only):

<A> And
Do a bit-wise AND of Value1 with Value2; result has all bits that are set in both Value1 and Value2.

<N> Not
Do a bit-wise negation of the current entry.

<O> Or
Do a bit-wise OR of Value1 with Value2; result has all bits that are set in Value1 or Value2.

<X> Xor
Do a bit-wise eXclusive OR of Value1 with Value2; result has all bits that are set in Value1 or Value2 but not both.

<L> shL
Shift Value1 left by Value2 bits. If Value2 is greater than 31 or less than 0, the result is 0.

<R> shR
Shift Value1 right by Value2 bits. If Value2 is greater than 31 or less than 0, the result is 0.

Other commands:

<K> assign current value to a Key
Convert the current value into an SMACS macro. (This command requires that SMACS be loaded.) When in a real number mode, you will be asked if you want the string of characters to be in the form of a typed constant array--for example, if the current value (line 2 of the calculation window) were equal to Pi, the string constructed by PCALC would be either '3.14159265358979324' (regular format) or '($40,$24,$93,$97,$58,$53,$26,$59,$41,$31)' (BCD typed constant format). This

command is particularly useful when writing programs using the TPBCD unit, since the Turbo compiler will not allow you to assign a constant value to a BCD real.

<Esc> Exit
Exit from PCALC.

<AltU><AltU> Unload
Unload PCALC from memory if it is safe to do so.

Other keys that can serve a function:

<BkSp>
Delete the last digit or character entered.

<F5>, <F6>, <F7>, <F8>, <F9>, <F10>
When in hexadecimal mode, these keys can be used to enter the values $A through $F, respectively.

<CtrlA>, <CtrlB>, <CtrlC>, <CtrlD>, <CtrlE>, <CtrlF>
When in hexadecimal mode, these keys can be used to enter the values $A through $F, respectively.

<.>
Insert decimal point (real modes only).

<E>
When in one of the two real number modes, <E> can be used to denote the start of an exponent. If <E> has already been used once for this purpose, it is construed to mean "clear Entry" if pressed a second time.

<->
If no numeric digits have yet been entered, <-> can negate the number about to be entered. Also, if an exponent is being entered (the last character in the current entry is <E>), <-> can be used to signify the sign of the exponent. Note that negative numbers may not be entered when in binary or hexadecimal modes, or when the current operation involves the shifting of bits to the left or right.

<+>
If an exponent is being entered (the last key pressed was <E>), <+> can be used to signify the sign of the exponent. This is unnecessary, however, since the exponent is assumed to be positive if no sign is specified.

Notes on switching calculation modes:

Although it is usually possible to switch from an integer mode to a real mode or vice versa, there are some restrictions. First, you cannot switch from an integer mode to a real mode if the current operation is not supported in real mode (Mod, Or, Xor, etc.). Second, you cannot switch from a real mode to an integer mode if any of the numbers in the calculation window is outside the long integer range (-2,147,483,648 to 2,147,483,647). Finally, when switching from a real mode to an integer mode, fractional numbers will always be rounded to the nearest whole number. This means that numbers between 0 and 1, such as 0.25, will be converted to either 0 or 1. It also means that results calculated in a real mode will often appear to be incorrect after you switch to an integer mode. For example, if the expression is "0.49 + 0.49 = 0.98" and you switch into Decimal mode the calculation will appear as "0 + 0 = 1" (everyone knows that...). Switching back into a real mode will not correct the display.

C. PTIME: Programmer's Time Management Utility

Most professional programmers need to keep track of the time that they spend on each project. But doing so can sometimes be difficult when you've got several projects going at once, especially if you get interrupted frequently. If this problem is yours--and perhaps even if it isn't--you may find PTIME a useful tool.

PTIME allows you to keep accurate records of how you spend your time. You can toggle timing on and off quickly and easily, use the pause feature to account for interruptions, save and load data files, set an alarm, activate an on-screen clock--you can even schedule an SMACS macro for playback while you're away from your desk. And because PTIME is memory resident, you don't need to stop what you're doing to use it.

Once PTIME is loaded into memory, you can activate it at almost any time by pressing <Ctrl><RightShift><T>. When you do, you'll see a display that looks something like this:

```
PTIME: Programmer's Time Management Utility 5.00
Alarm: 12:00 Macro: 00:00 Start: 09:07 Elapsed: 01:42 Time: 11:03:50
File: C:\TURBO\CURPROJ.DAT          Timer: ON  Pause: ON  Clock: OFF

[ESC] exit  Alarm  Clock  Disable  Load  Macro  Pause  Timer  Update
```

The top line of the display shows you, from left to right, the time at which an alarm is scheduled to go off, the time at which a macro is scheduled to be played back, the time at which you started timing your work, the amount of time that has elapsed since then (less any time during which the Pause feature was activated), and the current time. The second line shows you the name of the current data file, and the status of the Timing, Pause, and on-screen Clock toggles. The bottom line shows you a menu of available commands, described below.

Alarm

When you press <A>, you will be asked to

```
Enter the time in "HH:MM" format (HH = 0-23, MM = 0-59):
```

If you wanted to schedule an alarm for 1 p.m., for example, you would enter

```
13:00
```

To cancel the alarm before it is set, simply press <Esc> when prompted for the time. If the alarm has already been set, it can be cancelled later by pressing <A> a second time and entering <Y> to confirm that it should be cancelled.

Clock
Pressing <C> activates the on-screen clock feature. When you leave PTIME, the current time will then be displayed roughly three times a second. To turn off the clock, pop up PTIME and press <C> a second time. Note that the on-screen clock is always disabled temporarily while PTIME is popped up.

Disable
This feature allows you in many cases to unload PTIME from memory without rebooting the system. Because doing so is potentially dangerous, PTIME will not disable itself if any program loaded above it uses the same interrupt vectors.

Load
When you press <L>, you will be asked to enter the name of a file in which to store timing data. If it does not exist, a new data file will be created. Otherwise, PTIME will simply store the name of the file for future reference. This command is not available while the timer is on.

Macro
This feature allows you to schedule an SMACS macro for playback. If SMACS is not loaded when you press <M>, an error message will be displayed. Otherwise you will be asked first to press the key corresponding to the macro, then to enter the time at which the macro is to be played back. For example, if you wanted to play back a macro associated with <AltP> at 5:30 p.m. you would press <AltP> and enter

```
17:30
```

As with the Alarm command, you can cancel the playback of the macro before it has been scheduled by pressing <Esc> when prompted for the time, and you can cancel it later by pressing <M> a second time.

Pause
The Pause command allows you to account for brief interruptions in your work. If the timer is on and you get interrupted, you can temporarily deactivate the timer by pressing <P>. Although the timer will remain on, the elapsed time will remain unchanged until the Pause toggle is pressed a second time.

Timer
This is the most fundamental of the PTIME commands. When you press <T>, the program turns on its timer to begin tracking elapsed time and appends a new record to the current data file. If no data file has yet been selected with the Load command you will be prompted for one. To turn off the timer, press <T> a second time. The last record in the data file will then be updated.

Update

The Update command exists mainly to allow you to recover from any system crashes that might occur while the timer is on. If this happens, you can simply reload PTIME after rebooting, pop up the program, press <U> for Update, and enter the name of the data file. After PTIME has updated the last record in the file, you can then resume timing by issuing the Timer command. The only data that could be lost due to a system crash is that concerning the duration of Pauses in your work. Since that is the case, you may wish to use Update at the end of each Pause. Note that you can also recover from crashes manually by editing the last record in the data file with the PTRPT utility, described in the next section.

D. PTRPT: Report Generator for PTIME

PTRPT is a utility for generating reports from the data files produced by PTIME. It also allows you to examine, edit, append, and delete records in the data file, as well as to calculate earnings if you are using PTIME to track billable time.

When you load PTRPT, you'll see a menu that looks something like this:

```
PTRPT: Report Generator for PTIME Data Files 5.00
Append Browse Calculate Delete Edit File Print Quit Report Totals
```

The following options are available from this menu:

Append
Add a new record to the end of a data file. If a data file has already been selected, a new record will be appended to it immediately. You will then be given a chance to edit the record. See the discussion of the Edit command.

Browse
Browse through the records in a data file. This command puts you in Browse mode, in which you can examine all records in the data file. <PgUp> and <PgDn> move you backward and forward through the records, respectively. <Home> moves you to the first record in the file, and <End> to the last record.

Calculate
Calculate earnings based on hours worked. If no data file has yet been selected, you will be asked to specify one. You will then be prompted for an hourly rate to use in calculating earnings. To cancel the command, press <Esc>. After a rate has been entered, your total earnings will be calculated based on the cumulative amount of elapsed time for all records in the data file.

Delete
Delete records from a data file. When you give the Delete command, you will enter Browse mode (see the entry for the Browse command). Use <PgUp> or <PgDn> to locate the record to be deleted, then press <Enter>. Since deletion is an irreversible process, you will be asked to confirm that the record should be deleted. You can continue to delete records in this fashion until all records have been deleted or the <Esc> key is pressed.

Edit
Edit a record in a data file. When you give the Edit command, you will have access to all records in the database. Use <PgUp> or <PgDn> to locate the record to be edited. The <Up> and <Down> keys can be used to select a particular field for

editing. There are seven fields in each record that can be edited. Six of these will probably seem familiar and self-explanatory: 1) the date when timing was started, 2) the starting time, 3) the date when timing was stopped, 4) the stopping time, 5) the elapsed time, and 7) the optional comment. Basic instructions for editing these fields are displayed at the bottom of the screen. The sixth field, 'Allow Update', should almost always be No. This field tells PTIME whether or not the last record in a data file can be updated with the Update command. (See the documentation for PTIME.) PTIME sets it to Yes (True) when the record is first created, and to No (False) when timing is turned off. After editing a field, press <Enter> to move to the next field. <Esc> cancels all your edits to a database record. For a complete list of available editing commands, see the documentation for the TPENTRY unit.

File
Select a new data file. You will be prompted for the name of a valid PTIME data file. If the file is found and it is valid, all subsequent commands will operate on this new file. You can cancel this command by pressing <Esc>.

Print
Generate a report and send it to the printer. You will be prompted for the name of the data file, if one hasn't been selected already, and for an hourly rate. If you don't wish to specify a rate, press <Esc>. The report sent to the printer will show each record in the data file, formatted as follows:

```
Record #23
Start: 11/15/87 at  9:47:30 am
Stop:  11/15/87 at 10:21:21 am
Elapsed:  0:29:30  Earnings: $17.21
Modified MYPROG to support EGA and VGA.
```

The first line has the record number. The second line shows when timing started, and the third line shows when timing ended. The fourth line has the elapsed time and, if a rate was specified, the earnings for that particular work session. The fifth line has your comment, if any. The end of the report shows the total elapsed time for all records in the data file, and the total earnings (if a rate was specified).

Quit
Quit PTRPT. If a data file is open, you'll be asked to confirm that you want to quit.

Report
Generate a report and send it to a disk file. Report is identical to Print, except that it routes the output to a file rather than to the printer. The file created is a standard ASCII text file, so you can modify it using any text editor or word processor.

Totals
Display total number of records in the data file and hours worked. Totals is similar to Calculate, but does not show an earnings figure.

PTRPT also provides mouse support, using the facilities in TPMENU, TPPICK, and TPENTRY. See the documentation for those units for details.

E. PREF: Programmer's Quick Reference Chart

PREF is a memory resident quick reference chart for PC programmers. Taking the idea of SideKick's ASCII chart a few steps further, PREF displays a complete series of translation tables for values in the range 0-255. These tables

- show the displayable character corresponding to the value
- show the ASCII code for the value
- show the value in decimal, hexadecimal, and binary notation
- show the keys associated with the value--for example, 16 = <CtrlP> or <AltQ> (extended scan code)--including those keys that are available only if the TPENHKBD unit is being used (displayed in high intensity or white), as well as the pseudo-scan codes returned by certain routines in the TPMOUSE unit (displayed in high intensity or yellow)
- show the video attribute corresponding to the value for both monochrome and color display adapters (color samples included)

To activate PREF, press <Ctrl><RightShift><R>. Once PREF has been popped up, the following commands are available:

<Up>, <CtrlE>, <CtrlW>
Scroll up one line.

<Down>, <CtrlZ>, <CtrlX>
Scroll down one line.

<PgUp>, <CtrlR>
Scroll up a page.

<PgDn>, <CtrlC>
Scroll down a page.

<CtrlPgUp>, <CtrlQ><R>
Top of the chart.

<CtrlPgDn>, <CtrlQ><C>
Bottom of the chart.

<Home>, <CtrlQ><S>
Top of the chart.

<End>, <CtrlQ><D>
Bottom of the chart.

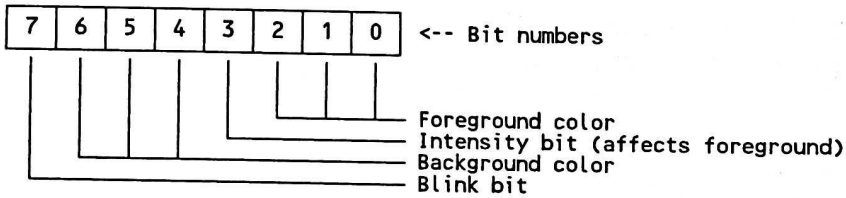
<Esc>
Exit from the program.

<AltB>
Toggle attribute blinking on and off. If attribute blinking is off within PREF, it will be reenabled when you exit from the program.

<AltU><AltU>
Unload PREF. Note that PREF, like the other memory resident demo programs, will not unload itself unless it is safe to do so. If it isn't, PREF will return you to the previous program, but it will remain resident.

Although PREF's display is pretty much self-explanatory, one peculiarity of it requires comment. But first some background...

Video attributes have four components, as this diagram shows:



Normally, any video attribute with a value greater than 127 (01111111 in binary) will produce blinking text. Since blinking text can be quite annoying if not used in moderation, PREF suppresses blinking in all of its color samples. To indicate attributes that produce blinking, it instead highlights the names of the foreground and background colors.

If you press <AltB>, however, blinking will be disabled at the hardware level, and bit 7 (the blink bit) will be used as an intensity bit for the background color. This allows for a broader range of color combinations than is normally possible.

PREF also allows you to use a mouse to move from one part of the chart to another. If a mouse is installed, PREF uses the far left edge of the chart to display a "scroll bar". The body of the scroll bar is a series of polka-dot blocks. The relative position of the attribute at the top of the screen within the chart is designated by a solid block in a different color. Immediately above and below these blocks are arrows, one pointing up and the other pointing down. To scroll the display down a bit, you would move the mouse cursor to the arrow pointing down and click the left mouse button. To scroll up, move it to the arrow pointing up and click the left button. To move rapidly to another section of the chart, move the mouse cursor to one of the polka-

dot blocks and click the left button. Clicking the right button does the same thing as pressing <Esc>, no matter where the mouse cursor is located.

Be forewarned that, in order to use the mouse in a TSR without affecting other programs, PREF (actually TPMOUSE) must use mouse functions that are not implemented in all mouse drivers (MicroSoft added these functions only when they needed them themselves for CodeView). Unfortunately, there is no way for the software to determine with certainty whether or not a particular mouse driver implements these functions. So if you encounter problems using your mouse in PREF, you either need to obtain a newer version of your mouse driver, or disable the 'UseMouse' directive in TPDEFINE.INC and recompile PREF.PAS.

F. MAKEMENU: Menu Maker for TPMENU

MAKEMENU is an interactive generator of hierarchical menu systems. With it, you can visually develop such systems, quickly and easily achieving proper positioning, colors, and appearance that would otherwise take much longer to write in Pascal. After you are satisfied, MAKEMENU will generate Pascal source code or write a binary menu file to store one or many menu systems.

When you type MAKEMENU from the DOS command line, you'll see a menu similar to the following:

```
Turbo Professional Menu Maker 5.00
Global          Submenu          Item                File

                                 Add      ^I   <——
                                 Edit     ^E
                                 Move     ^A
                                 Order    ^O
                                 Key
                                 Help
                                 Select Char
                                 Remove
```

Since you initially have no menu system defined, the first thing you'll probably want to do is to add a menu item. Hence, MAKEMENU selects the Item/Add command for you. If you press <Enter> at this point, MAKEMENU will prompt you for the name of the item to add. If this is your first time using MAKEMENU, though, don't press <Enter> yet, but follow along with this documentation for a few more minutes.

When you browse around MAKEMENU by pressing the cursor keys, <Enter>, and <Esc>, you'll notice that you are not allowed to select certain items at this time. That's because you don't yet have a menu system, and some of the commands don't make sense until you do. Once you've added at least one item to the menu, or read in a menu from a disk library, MAKEMENU will enable the rest of the commands for you.

Wherever you are in the menu system, MAKEMENU displays a help line at the bottom of the screen. You'll also note that many of MAKEMENU's commands are associated with a hot key (shown at the right edge of the menu, and in the right-hand column of the descriptions to follow). For example, you can access the Global/Choose command from anywhere in the menu system by pressing <CtrlC>. Once you've used MAKEMENU for a while, you'll appreciate the efficiency of using the hotkeys for common functions.

If you have a mouse (Microsoft compatible), you can also use it in MAKEMENU. Pressing the left mouse button is equivalent to pressing <Enter> on the keyboard.

Pressing the right button is equivalent to pressing <Esc>. Pressing both buttons is equivalent to pressing the <Space> bar, but only when you are using the Global/Choose command, described below. By moving the mouse around, you can select items from either MAKEMENU's menu or from the one you're designing. The mouse also can drag menus or items around on the screen--what it does will be intuitive and obvious when you use it.

Before we describe all of MAKEMENU's commands, let's step through a short session, building a menu system from scratch. To get started, select the Item/Add option again and press <Enter>. MAKEMENU prompts for the name of the item. Let's enter 'Data', followed by pressing the <Enter> key. (Don't type the quote marks when entering Data.) MAKEMENU now prompts you for a help line to go with the item, just like the help lines that appear on the bottom of MAKEMENU's screen. For now, just press <Enter> at this prompt to proceed without adding help.

You'll now see 'Data' in a framed box near the top left corner of the screen. You'll also notice that all of MAKEMENU's Item commands are now enabled, and that you can move the cursor freely around this submenu. Let's add a second menu item called 'Print'. Assure MAKEMENU's highlight bar is positioned over Item/Add and press <Enter> again. Respond to the prompts for item name and help line. Now you'll see that the menu box has grown and that 'Print' is positioned beneath 'Data' within the box.

Hmm, we really want this top level submenu to scroll horizontally and not vertically. Let's change the orientation. To do so, press the <Left> arrow key once to move to MAKEMENU's Submenu commands. Move the cursor down to Submenu/Settings and press <Enter> to select the nested menu. Now move the cursor down to Orientation, press <Enter>, then to Horizontal and press <Enter>. Your menu at the top of the screen shifts orientation, the frame is resized, and now 'Data' and 'Print' appear side by side.

While we're here, let's change another setting. Press <Esc> once to get back to the Settings menu, move the cursor to the Shadow selection, and press <Enter>. Now your menu has a shadow around it. You can pick any of the selections from the Settings menu and by pressing <Enter> immediately change the appearance and properties of current submenu you're building, without affecting any other submenus.

Now let's add a nested submenu, attached to the 'Print' item. To do so, press <Esc> to get out of the Settings menu and move the cursor up a few lines to the SubMenu/Add command. Press <Enter> here to add a nested submenu. You'll immediately see a small box appear below 'Print'.

Now we can start adding items to the new submenu. Let's use a hotkey this time to select the Item/Add command. To do so, press <CtrlI>. MAKEMENU's menus don't change because we want to leave the current context intact. Even so, MAKEMENU now prompts for the name of the new item. Enter 'Options' and again leave the help line empty. Press <CtrlI> to add yet another item, and enter 'Go' for this item name.

It's convenient that MAKEMENU is still on its Submenu commands, since we'd like to adjust the new submenu so that it meshes well with its parent. Move the highlight bar down to Settings again, press <Enter> and select the Frame command. Move the highlight down to item 4 on this menu, press <Enter> and now your submenu fits nicely with the one above it.

What we want to do next is add a submenu below the 'Data' entry. To do so, we must use another MAKEMENU command, the Choose command. To activate the Choose command, press <Esc> a couple of times, until only the MAKEMENU main menu is showing, and then press the <Left> arrow once to get to the Global submenu. Press <Enter> to pull down the menu, then <Enter> again to activate the Global/Choose command.

This command lets you use the menu system you're developing just as if it has been implemented in a program. You can use the cursor keys, <Enter> and <Esc> to browse around all of the menu system that's been entered so far. When you get to a menu item that you'd like to modify in some way, press the <Space> bar to return control to MAKEMENU. In this case, assure that the highlight bar is over 'Data' and then press <Space>.

With 'Data' selected, go back to MAKEMENU's Submenu/Add command and press <Enter>. You'll see a new little submenu appear below 'Data'. Now follow the same sequence of actions you used to add items to the previous submenu: press <CtrlI> for Item/Add, enter 'Read', then <CtrlI>, and enter 'Edit'. These items will appear in the submenu as you add them. Now you can use the Submenu/Settings/Frame command again to adjust the frame of this submenu to fit its parent.

You'll note that the submenus we just added don't have shadows, even though we modified the top level menu to have a shadow. That's because when we adjusted the top menu, we used a command that affected only that one submenu. To change the *default* properties of the menus, you should use MAKEMENU's Global/Defaults commands, which otherwise are very similar to the commands for Submenu/Settings. Let's not deal with that right now, although you may want to remember in the future to adjust Global/Defaults to your needs before starting to design a new menu system.

To finish this example session, let's save the menu we've designed to a disk library, and also generate source code for it. Press <Esc> until you are at MAKEMENU's top level, then the <Right> arrow until the highlight bar marks File, then <Enter> to select the submenu. Move the cursor down to the Write lib item and press <Enter>. MAKEMENU will prompt you for a file name. Enter the name of any non-existent file, remembering that MAKEMENU applies a default extension of .LIB to whatever name you enter. Next MAKEMENU will prompt you for a 'Menu library ID'. This is any short string (up to 12 characters) that distinguishes this particular menu system from any others that may reside in the same library. You'll need to know this ID string when you recall a menu system from the library, so pick a mnemonic name. In this case, let's enter 'TEST1'. MAKEMENU quickly writes a binary image of your menu system to the library, and then asks you to press <Esc> to continue.

Since the menu has been written to disk, there's no need to do so again, so MAKEMENU disables the Write lib command until you make another change. You can, however, still generate Pascal source code for the menu system, and MAKEMENU conveniently positions the cursor over that item for you. Press <Enter> here and MAKEMENU will prompt you to enter a file name. Enter the name of any non-existent file here, remembering that MAKEMENU applies a default extension of .PAS to whatever name you enter, and that it will overwrite any file of the same name. Let's enter 'TEST1.PAS' here. MAKEMENU writes the source code in a flash, and asks you to press <Esc> to continue.

Remember, if you expect to do more editing on a menu system, always save it to a menu library before you exit MAKEMENU. Generating source code is not enough! MAKEMENU can read a menu from a library so you can edit it again, but it can't read source code. MAKEMENU will always warn if you try to quit after making changes without saving them to a library.

That's enough to get you started, so move the highlight bar down a couple of lines to the File/Quit command, press <Enter> and MAKEMENU will return you to DOS. You can immediately compile TEST1.PAS, run it, and take another look at the menu system you've just designed. With MAKEMENU, you'll be amazed at how fast you can develop full-featured, good-looking menu systems!

The remainder of this section will serve as a reference to all of MAKEMENU's commands.

Global

Choose **<CtrlC>**
Erases MAKEMENU's own menu system and lets you browse around the menu

system you're building. Use the cursor keys, <Enter> and <Esc> just as you would in the final program. When the highlight bar selects an item you'd like to modify using MAKEMENU, press the <Space> bar.

New **<CtrlN>**

Clears the current menu system and initializes a fresh one. Will warn if your current work has not been saved to a library. There is no need to use this command when you first start MAKEMENU.

Defaults

Brings up another submenu showing the various menu defaults. Changing these defaults will have two effects. First, the settings you provide will be applied to all further entries that you make during the MAKEMENU session. Second, when you change any default setting, that change is applied to the current menu system, starting with the current submenu and continuing to all of its children.

Where it makes sense, MAKEMENU displays the current default value beside the menu item. MAKEMENU has a set of "factory" defaults. When you read a menu in from a library, however, MAKEMENU adjusts the defaults to match those of the root menu of the library.

The following items appear on the Defaults submenu:

```
Colors
Frame
Help row
Orientation
Explode
Shadow
Delay
Map
Restore
```

Colors

Provides a nested menu showing the color types to modify. These include colors for Frame, Title, Body, Selected, Pick, Help, Disabled, and Shadow portions of the menu system. "Body" is the color for items not currently selected. "Pick" is the color for the one letter within each deselected item which, if pressed on the keyboard, selects the item. "Selected" is the color for each selected item. "Disabled" is the color for temporarily disabled items that cannot be selected by any means. When you select one of these colors to change, MAKEMENU displays a color selection box. The current attribute value in hex is displayed at the top of this box. Use the cursor keys to move the reversed diamond to the new color, and press <Enter> to select it. Press <Esc> to quit without changing the color.

Frame
Provides another nested menu, showing the various frame types for submenus. If the frame you want is not available, choose one that's close and modify the source code later.

Help row
Prompts for the row where a help line will appear while your menus are selected. If you don't want a help line, enter zero. Doing so will prevent the Item/Add command from prompting for a help string each time.

Orientation
Provides another nested menu, so you can select between horizontal and vertical menu scrolling.

Explode
Toggles between exploding and non-exploding menus.

Shadow
Toggles between shadowed and non-shadowed menus.

Delay
Prompts for the delay per stage (in milliseconds) between stages of explosions. This is the only location within MAKEMENU where you can change this value, since explosion delay is a global property of each menu system.

Map
Controls whether MAKEMENU applies color mapping to the menu you're designing. With color mapping On, and when the video system is in monochrome (MODE MONO) or composite black and white (MODE BW80), the menu system maps colors to a restricted set that improves visibility in those modes. Selecting this item toggles the Map mode.

Restore
Restores all default values to their factory settings. Unlike the other global defaults commands, Restore does not immediately modify your current menu system to reflect the new defaults.

Redraw
Refreshes the screen if it has gotten corrupted by previous menu operations. This should almost never happen.

Info
Displays the amount of free heap space available to MAKEMENU.

Keys
Completely renumbers the key numbers assigned to the items of the current menu system. Use this command with caution if you're modifying a menu system which you've already integrated into a program, since it will potentially change key numbers for items you've already referred to.

A little background on this command: each item of a menu system must have a unique key value for identification. Typically these keys are small integer values that you reference in a case statement to decide what action to take after a user has made a menu choice. When you build a menu system using MAKEMENU, it automatically assigns key values to each item you add, starting with 1 for the first item, and using the next available integer for each item thereafter. If you modify a menu system, or if you enter it in a haphazard fashion, the key numbers won't follow any obvious pattern. This is when the Global/Keys command is handy: it assigns the key numbers using a first-item-to-last, depth-first manner which causes the items to appear in numerical order when MAKEMENU generates the source code. It's probably a good idea to use the Global/Keys command when your menu design has stabilized, just before you're ready to integrate it into a program.

Submenu

Add **<CtrlS>**
Attaches a nested submenu to the current item. You can do this only once per item. When you do so, MAKEMENU will draw a small box that shows you where the submenu will start. (For "no frame" submenus, the box will be invisible unless the background of the "body" color isn't black.) The submenu will grow automatically as you add items to it, or you can use the Submenu/Resize command to change it manually.

Drag **<CtrlD>**
Repositions a submenu and any menus nested under it. You won't see the nested submenus move, but the next time you select them, you'll see the difference. Use the Submenu/Move command to move a submenu without dragging its children along.

Settings
Changes the properties of the current submenu without affecting any other menus or the defaults for further additions. Selecting the Settings command brings up another menu with the following options:

```
Colors
Frame
Help row
Orientation
Explode
Shadow
```

These options work just like those for the Global/Defaults command. See above for details.

Title
Lets you enter or edit the title of the current submenu. The title is displayed in the top frame of the menu and is allowed only when a frame is shown. TPMENU will truncate the title if it's longer than the width of the menu. Increasing the width (with the Submenu/Resize command) makes more of the title show.

Move
Repositions a submenu without affecting any menus nested under it.

Resize
Adjusts the size of the current submenu. MAKEMENU won't let you reduce the size of the menu below the minimum that surrounds all the items. MAKEMENU automatically resizes the submenu if the addition of an item requires it, and in some other situations: for example, when the orientation of the menu is changed.

Kill
Removes all descendants of the current item. Note that it does not remove the current submenu itself. When you use Kill, you'll get a chance to confirm the action. After Kill is done, you won't see any immediate change on the screen, since the nested menus aren't visible at the time anyway. Use the Item/Remove command to remove a menu item.

Item

Add **<CtrlI>**
Adds another item to the current submenu. You must add each item before you can edit, move, or perform any other operation on it. When you select this command, MAKEMENU will prompt you for the name of the item. Then, if help is enabled for the menu system, MAKEMENU will prompt for a help line to be displayed whenever the item is selected.

Each item is added just after the currently selected item. When you're entering a new submenu, the items will appear one after the other. Note, however, that you can also use this capability to insert new items in the middle of an existing submenu.

Edit **<CtrlE>**
Lets you change the name of an existing item.

Move **<CtrlA>**
Changes the position of an existing item. Note that you can shift the item in only one dimension--for horizontal menus, you can shift items left and right, for vertical, only up and down. Also note that moving an item does not change the order in which items are selected. Hence, it's a bad idea to move an item so much that it passes a neighboring item. For that purpose, use the Item/Order command instead. When you move items, MAKEMENU adjusts the menu borders so that they always surround every item.

Order **<CtrlO>**
Changes the order in which items are selected. Use this command in combination with Item/Move to shift menu items around and readjust their position.

Key
Modifies the default item key for the current item. MAKEMENU assigns a default integer key to each item as you add it. If that value isn't acceptable, change it using this command. MAKEMENU always displays the key value for the current item next to this command.

Help
Lets you enter or edit the help string to appear for the current menu item. The help string will be displayed on the row specified by Submenu/Settings/Help row for the current submenu. If that value is 0, the help string won't appear at all.

Select char
Lets you adjust the position of the selection character for the current menu item. When people use your menu system, they can press the selection character and quickly select the corresponding item of a submenu. As such, you'll probably want each item in a submenu to have a unique select char or else want to disable selection by character altogether. To disable, press the <Left> arrow until the selection highlight disappears. Otherwise, just use <Left> and <Right> to highlight the character you want to select the item.

Remove
Deletes the current item. You'll get a chance to confirm this operation. You can't remove an item until you've killed any submenus nested below it. Use the Submenu/Kill command for that purpose.

Read lib **<CtrlR>**

Reads a menu system from a library previously stored with the Write lib command. Enter the file name where the library is stored (typically has an extension of .LIB). You'll also need to know the menu ID of the particular system you want from the specified library file. This is a string up to 12 characters long, which you've previously specified when you wrote the library. Once you've entered these values, MAKEMENU reads and displays the top level of the selected menu system. It also modifies the Global/Default settings to match those found in the top level of the menu system just read.

Write lib **<CtrlW>**

Writes the current menu system to disk. Enter the file name where the library will be stored. MAKEMENU applies a default extension of .LIB to whatever filename you enter. You'll also need to enter a "menu ID". This is a string up to 12 characters long, which differentiates the current menu system from any other that may be stored in the same library file. If the menu ID you specify is the same as one already found in the library, the current menu replaces the one already stored there. Since TPMENU library files hold at most 10 menu systems, and since obsolete versions of a menu remain in the library until it is packed, you may run out of library directory space if you've modified the same menu system many times. In this case, use the File/Pack command to remove obsolete entries before writing to the library again.

Generate source **<CtrlG>**

Writes source code for the current menu system to disk. Enter the file name where the source code will be written. MAKEMENU applies a default extension of .PAS to whatever you enter. The generated source file is ready to compile, and contains enough code that you can exercise your menu system without further programming.

Be sure to note that MAKEMENU cannot directly read and interpret source code. If you expect to perform further modifications to a menu system, be sure to save it in library format as well as source code.

Pack lib **<CtrlP>**

Removes obsolete entries from a menu library. This is important if you've stored more than one version of the same menu system, using the same menu ID, to a menu library. Enter the name of the library you want to pack. MAKEMENU generates a temporary file name where it will write the packed output. When packing has been successfully completed, MAKEMENU deletes the original file and renames the temporary file back to the original.

Quit **<CtrlQ>**

Returns to DOS. If you've made changes since you last saved your work to a library file, you'll get a chance to change your mind.

G. BSORT: File Sorting Utility

BSORT is a fast, large-capacity replacement for the DOS SORT filter. Use it for sorting text files into a specified order.

BSORT is command line driven. Its syntax is as follows:

BSORT [Options] <InputFile >OutputFile

If you type BSORT by itself, it will provide a brief help screen and then halt. Acceptable options are:

/R	Sort in reverse order
/I	Sort ignoring case
/B n	Sort with key starting in column n
/L n	Sort with maximum key length of n characters

If BSORT is called without any of these options, it sorts in ascending order using a sort key spanning the entire length of each line. The options may be used separately or in combination.

Here are a few examples of using BSORT:

BSORT <TEST.DAT >TEST.SOR

Sorts TEST.DAT in ascending order, writing output to TEST.SOR.

BSORT /I /R <TEST.DAT >TEST.SOR

Sorts TEST.DAT in descending order, ignoring case.

DIR | BSORT /B 10 /L 3

Sorts a directory by extension.

BSORT works in either of two ways. If neither the /B nor /L option is specified, it keeps the entire text of each line in memory during the sort, then writes the output file directly from memory. Otherwise it stores only the specified portion of each line in memory during the sort. To write out each line it then seeks to the appropriate position in the input file and copies the full line from there to the output. Although the second alternative is slower, it conserves memory when large files are sorted.

The maximum allowable line length for input strings is 255 characters. You could use the TPASCIIZ unit to enable longer lines if needed.

H. TFIND: Text Find Utility

TFIND is a handy utility for managing the large number of files that programmers always create. It searches a disk for files by name or by contents.

TFIND is command line driven. If you type TFIND alone, it will write a brief help screen and then halt. Here is the full syntax:

TFIND [Options] [MatchString] [>OutputFile]

MatchString specifies a string of text that TFIND should seek. If no MatchString is given, TFIND will report all files found consistent with the options.

/C	Make searching case sensitive
/F	Show matching file names only
/N	Exclude no file extensions from search
/R	Recursively search all subdirectories
/S [d:][path\][mask]	Start search in d:path and search only files matching mask

If no options are given, TFIND will search all non-excluded files in the current directory to find MatchString. Because TFIND is designed to search text files, it automatically excludes files with certain extensions: COM, EXE, OBJ, TPU, TPL, TPM, BIN, WKS. You can add to this list by modifying a constant in TFIND.PAS and recompiling the program.

Whenever TFIND finds a match it writes the file name, line number and matching line to the standard output device. By default, TFIND ignores case when searching.

When the /F option is specified, TFIND reports each file containing a match. It writes the file name just once with no line number or text line.

When the /N option is specified, TFIND searches binary files as well.

When the /R option is specified, TFIND searches not only the current directory but also any subdirectories of the current directory.

By using the /S option, you can tell TFIND to start searching in another directory besides the current one. You can also specify DOS wildcards to limit searching to a certain class of files.

The TFIND MatchString offers considerable flexibility. For any single word composed of normal ASCII characters you will simply type the word. If the match contains several words separated by white space, the match string should be surrounded by quotes (either single or double quotes are acceptable). If the match

must specify control characters or extended ASCII characters, use the following notation: #nnn, where nnn is a decimal number that specifies the ASCII code for the character.

Note that TFIND matches cannot spans multiple lines. In TFIND, maximum line length is 1024 characters.

Here are some examples of TFIND commands:

TFIND /C /S *.PAS FastWrite

Searches all PAS files in the current directory to report lines containing the word FastWrite. Searching is case sensitive.

TFIND /R /S C:*.BAT >ALLBAT

Makes a list of all BAT files in every directory of drive C:. The list will be stored in a file named ALLBAT.

TFIND /F /N #129#14AuxOut

Searches all files (including binary ones) in the current directory for the character string composed of an ASCII 129 followed by ASCII 14 followed by the word AuxOut.

TFIND "Don't do that"

Searches all text files in the current directory for the phrase "Don't do that". Note that single apostrophes may be embedded in a match string that is delimited by double quotes, and vice versa.

I. REP: Command Repeater

REP is a program modeled after one of the same name in the TurboPower Utilities. After taking advantage of the Turbo Professional libraries, REP became much smaller and simpler! With REP you can automate many tedious and repetitive tasks. It automatically runs another program with variations specified by you. REP is often used in combination with the output of a directory or file finder program to perform an operation on a group of related files.

REP is command line driven. If you type REP alone, it will write a brief help screen and then halt. Here is the REP command syntax:

REP "Command" [Options] [<InputStream]

Command is the program, batch file, or DOS internal command that you'd like to repeat. If Command contains more than one word, it must be surrounded by quotes. Command may not directly specify input redirection, output redirection, or piping. The presence of the <, >, or | characters within quotes confuses DOS, which will interpret them to mean that REP's input or output is redirected instead of Command's. See the later section on parser symbols for a way to specify redirection of Command.

REP expects to read an input stream that tells REP how to customize each repetition of Command. The input stream is read from the standard input device. Often REP's input will be piped in from the output of a directory lister. When no input stream is specified, REP will simply repeat the same command one or more times. This behavior is useful in combination with other REP options, described below.

Here are the valid options for REP:

/K KeyList	Pass keys in KeyList to Command
/Q	Query before proceeding
/R n	Repeat Command n times

The /K option tells REP that it should send a stream of keystrokes to the application it is executing. This is often useful to provide certain "boilerplate" responses, such as answers to the question "Are you sure?". The KeyList may contain normal text or characters specified by a numeric sequence #nnn, where nnn is the decimal ASCII value of the desired character. Specify extended keys with a #0 followed by the scan code of the desired character. For example, the <F1> key is written with the sequence #0#59.

Use the /Q option if you wish to preview each of REP's commands before it is executed.

Use the /R option to simply repeat a given command n times. This option is valid only when an InputStream is not provided. If no InputStream is offered, and the /R option is not specified, REP runs Command one time. The latter behavior is useful when REP is being used primarily for its ability to pass keystrokes to an application.

How does REP utilize information from the input stream? It supports what we call "parser symbols." Each parser symbol will refer to some portion of each input line, and REP will replace the parser symbols wherever they appear in Command or KeyList with the actual data.

Here are the parser symbols that REP supports:

@n	nth word in the input line (for n in 1..9)
@0	whole input line
@B	discretionary DOS backslash \
@D	directory name from first word of input line
@F	filename from first word of input line
@I	input redirection symbol <
@O	output redirection symbol >
@P	piping symbol |

REP divides each input line into words separated by spaces or tabs, just like the ParamStr routine in Turbo Pascal. You can refer to the first word with @1 or the ninth word with @9. Specify the entire input line with the symbol @0.

The @B, @D, and @F symbols are useful for parsing pathnames. They work especially well with the output of the TFIND program supplied with Turbo Professional. @D returns just the directory portion of a full path. @F returns just the filename portion. @B can be used to glue the two together again. @B is replaced with a backslash if the string preceding it does not end in backslash; otherwise @B is replaced with an empty string.

The @I, @O and @P symbols are used to specify redirection in Command. These symbols are replaced with the standard DOS characters prior to executing Command.

That's enough on the theory of REP. Some examples should make it clear.

REP "DEL *.*" /K Y#13

Deletes all files in the current directory, automatically answering the prompt "Are you sure?".

TFIND /R /S *.BAK | REP "DEL @0" /Q

Finds all BAK files in all directories of the current disk and deletes them after confirmation.

DIR *.PAS | REP "DIFF @1.PAS @1.BAK @O@ODIFFS"

Finds differences between each Pascal file in the current directory and its backup file, appending all results to the file DIFFS.

TFIND /R /S *.* | REP "COPY @0 A:\@F"

Copies all files in the current directory and any subdirectories to the root directory of drive A:.

J. DIFF: Difference Finder

DIFF compares two text files and reports the differences. It also serves as a tour-de-force demonstration of DOS memory management techniques, supporting normal memory, expanded memory and extended memory in one program. DIFF can also be compiled for DOS protected mode operation.

DIFF is command line driven. If you type DIFF alone, it will write a brief help screen and then halt. Here is the DIFF command syntax:

DIFF FileA FileB [/N] [>Differences]

DIFF's single command line option, /N, forces it to use normal memory for text storage as opposed to expanded (EMS) or extended memory. By default, DIFF checks all three forms of memory and uses the one with largest free space for storing text during the comparison operation. (Actually, if both expanded memory and extended memory are available, it will give priority to expanded memory, since it can be accessed more rapidly.)

DIFF writes any differences it finds to the standard output device. It reports them in the following format:

```
****Delete lines 45-46 of T1
Makes a list of all BAT files in every directory of drive C:. The
list will be stored in a file named ALLBAT.
****Insert lines 45-46 from T2
Searches all files (including binary ones) in the current
directory for the character string composed of an ASCII 129
```

A descriptive comment beginning with '****' shows the file name and line numbers which differ. The descriptive comment is followed with the actual text which was deleted from 'FileA' or inserted from 'FileB'.

Note that DIFF compares lines exactly. Differences in case or white space are significant.

K. VBROWSE: Virtual File Browser

VBROWSE is a file browsing utility that uses TPVARRAY to allow files of almost any size to be viewed (provided you have lots of disk space). VBROWSE is not intended to "compete" with more efficient file browsers, such as LIST.COM. It reads entire files into memory before allowing you to browse through them, and it can create *very* large "swap files". For these reasons, you will probably not want to use it if you don't have a hard disk with lots of disk space available. The purpose of the program is primarily to illustrate the use of the big array routines.

VBROWSE is invoked as follows:

VBROWSE FileName [d:\path]

"FileName" is the file to be browsed. (VBROWSE will search DOS's PATH to find it, if necessary.) If no filename is specified, you will be prompted for one. "d:\path" is an optional drive and directory for TPVARRAY's swap file. This option allows you to use another drive (a RAM disk, perhaps) for temporary storage. For example,

VBROWSE vbrowse.pas d:

would instruct VBROWSE to load VBROWSE.PAS into memory, and to write its swap file to D:.

The following commands are available in VBROWSE:

<Left>, <CtrlS>
Scroll left one column.

<Right>, <CtrlD>
Scroll right one column.

<CtrlLeft>, <CtrlA>
Scroll left 20 columns.

<CtrlRight>, <CtrlF>
Scroll right 20 columns.

<Up>, <CtrlE>, <CtrlW>
Scroll up one line.

<Down>, <CtrlZ>, <CtrlX>
Scroll down one line.

<PgUp>, <CtrlR>
Scroll up a page.

<PgDn>, <CtrlC>
Scroll down a page.

<CtrlPgUp>, <CtrlQ><R>, <Home>
Top of file.

<CtrlPgDn>, <CtrlQ><C>, <End>
Bottom of file.

<CtrlQ><F>
Find string.

<CtrlL>
Repeat last find operation.

<CtrlQ><L>
Goto line.

<CtrlK>n
Set marker n (0..9).

<CtrlQ>n
Goto marker n (0..9).

<AltP>, <CtrlK><P>
Print lines

<AltR>
Toggle ruler line on/off.

<F3>, <CtrlK><D>, <AltF>
Load new file.

<Esc>, <CtrlBreak>
Exit from the program.

<F1>
Display a list of the available commands.

VBROWSE also provides a limited degree of mouse support (at least in comparison

to MAKEMENU, PTRPT, PREF, and ENTRY). While browsing through the file, the following mouse commands are available (assuming that a mouse is installed):

<Right Button>
Exit from program.

<Both Buttons>
Display a list of the available commands.

When editing a string using the line editor (from TPEDIT) or when responding to a prompt asking you to press a key, the mouse buttons are assigned the following functions:

<Left Button>
Same as pressing <Enter>.

<Right Button>
Same as pressing <Esc>.

L. ENTRY: Data Entry Example

Although it doesn't do anything useful, the demonstration program ENTRY.PAS is worthy of special attention because it offers examples of how to use several of the units and capabilities in Turbo Professional. Using five of the new units--TPDATE, TPENTRY, TPPICK, TPMEMO, and TPHELP--it demonstrates, for example, how to do sophisticated data entry screens with full data validation; how to implement context-sensitive help systems; how to use pick lists to allow the user to select from a lengthy list of choices; how to perform background tasks while waiting for keyboard input; how to implement user-defined commands using a unit such as TPENTRY that processes keyboard input; and how to write programs that adapt to different currencies and date/time formats.

To run the demonstration program, you need two files found on the DEMO disk: ENTRY.EXE and ENTRY.HLP (which contains the data used by the help system). When you load the program, you'll see that the screen is divided into three sections. The top section is a box divided into two parts, the second of which serves as a status line, showing the current record number, the date, and the time. The date and time are kept current by a user-defined keyboard input routine which displays them as a background task. The middle section is a data entry screen consisting of several fields, which we'll discuss in a moment. The bottom section is also a box divided into two parts. The first part is a help line that provides brief instructions on what should be entered in the current field; the help text changes as you move from field to field. The second part is a static help line that tells you what keys perform some of the basic functions.

The main entry screen, designed with TPENTRY, has fifteen fields. The list below draws attention to any special qualities of a given field.

- Name: Nothing very special about this field. If the field already contains text when you move to it, and the first key pressed is an ASCII character, the field will be cleared. If the field editor is in insert mode, text at the very end of the field can be pushed off the end. Press <Enter> or <Tab> to move to the next field when you're finished.
- Address: This field's only special characteristic is that it is marked as required, meaning that it cannot be left blank.
- City: Unlike the Name field, text at the end of the field cannot be pushed off the end while in insert mode.
- State: You have two choices here. You can either enter a two-letter abbreviation for a state name, or you can press <F2> and choose an abbreviation from a pick list created with TPPICK. Only alphabetical characters may be entered in this field, and if you enter only a single letter you'll get an error message.

- Zip: You can enter either a five- or a nine-digit zip code here. Only numbers and spaces may be entered in this field, and an error message will be displayed if you try to enter a zip code with something other than 5 or 9 digits.
- Phones: This is a dummy field that leads to a second data entry screen. To get to the second screen, displayed in a popup window, press <Enter>. The two fields in the entry screen are discussed below.
- Gender: This is a multiple choice field. To select from the various choices (Unknown, Male, Female), press either <Space> or <+> to go forward through the list of choices. The <-> key can take you through the list backwards.
- Married: This is a yes/no field. Press <Y> or <N>, or <Enter> to accept the default answer.
- Born: This is a date field. You can enter any date between 1/1/00 (1900) and 12/31/99 (1999) here. Special validation routines in TPDATE and TPENTRY insure that the date entered is a valid one, and you'll get an error message if you try to enter an invalid one (like 1/32/88).
- Age: This is a calculated field, so you can't change it. Using one of the hooks provided by TPENTRY, the value for this field is calculated and displayed as soon as an entry has been made in the Born field. This field is also right-justified.
- Hourly wage: This is a numeric field that uses an alternate field editor provided by TPENTRY. Numbers are entered from right to left in calculator fashion. The decimal point is fixed, and the dollar sign floats.
- Weekly: Like Age, this is a calculated field. Its value is equal to "Hourly wage" * "Hours/week". The decimal point is fixed, and the dollar sign floats. The field is padded with asterisks rather than spaces.
- Hours/week: This is a special kind of multiple choice field. Rather than toggling though a list of choices, you increment or decrement the default value using the <+> and <-> keys.
- Yearly: This too is a calculated field. Its value is equal to "Weekly" * 52. The decimal point is fixed, and the dollar sign floats. The field is padded with asterisks rather than spaces.
- Notes: This field pops up a memo window that uses the TPMEMO unit.

The second entry screen, accessed by moving the cursor to the Phones field and pressing <Enter>, has two fields.

- Work phone: Only numbers and spaces may be entered in this field. A valid entry will consist of a complete (seven-digit) phone number, with or without an area code. If you try to enter a one or two-digit area code, or an incomplete phone number, an error message will be displayed. The field is padded with underscores ('_') rather than spaces.
- Home phone: This field has the same characteristics as the Work phone field.

ENTRY is also notable for its context-sensitive help system, developed with TPHELP. As you move the cursor from one field to another, try pressing the <F1> key for details on that particular field. If you see a word or phrase highlighted in purple (on color screens) or reverse video (on mono screens); these are links to related help topics. Pressing <Enter> will display the help screen for that topic. You can then return to the previous topic by pressing <AltF1>. You'll notice that some help screens have more than one link: if you see a word or phrase displayed in a light blue color (on color screens) or one that is underlined (on mono screens), you can select that topic by moving the cursor to that word and pressing <Enter>. The help screen for the Hourly wage field, for example, has four links.

To see an index of all the help topics, press <F1> from anywhere in the help system. You can then choose any of the topics by moving the highlight bar to the appropriate topic and pressing <Enter>. A few of these topics (those at the beginning of the list) cannot be accessed except from the index. Among these are topics concerning the various editing and cursor movement commands that are available.

Although the help system briefly describes each of the available commands, there are a few worth drawing your attention to here:

<AltB>
Normally when you enter an illegal character, ENTRY will beep at you. To keep it from doing so, press <AltB>, which toggles the error bell on or off. This is another example of a user-defined command.

<Tab>, <ShiftTab>
These keys move the cursor to the next or previous subfield, which is not necessarily the same thing as the next field. The Born field, for example, has three subfields: month, day, and year.

<Up>, <Down>
These keys move the cursor up or down on the screen, *not* to the next or previous field.

<PgUp>, <PgDn>
Pressing these keys move you to the next or previous record. Although you can enter data for up to 20 records, keep in mind that ENTRY does not provide any way to save these records. Don't waste too much time filling in the various records.

<Esc>, <CtrlBreak>
This restores the previous contents of the current record and gives you a chance to abort the editing session.

<CtrlEnter>, <CtrlK><D>, <CtrlK><Q>
Pressing any of these keys ends the data entry session. Although you can't tell it, the previous contents of the current record are not restored as they are when you press <Esc>.

Thanks to the low-level facilities furnished by TPMOUSE, and the higher-level ones in several other units, ENTRY is also able to support the mouse in fine fashion. While editing fields, you can do the following things using a mouse (if one is installed):

- Move the mouse cursor (shaped like a diamond) to any unprotected field on the data entry screen and select it by clicking the left mouse button
- Move the mouse cursor to the Phones field and click the left mouse button: the secondary entry screen associated with it will be popped up
- Move the mouse cursor to the State field and click the left mouse button once to select it, a second time to pop up the pick list of state names
- Move the mouse cursor to the Marital status field and click the left mouse button once to select it, a second time to toggle the value
- Move the mouse cursor to the Age or Hours/week fields and click the left mouse button once to select the field, a second time to increment the value
- Press the right mouse button to simulate pressing <Esc>
- Press both mouse buttons (the left and the right) to bring up the help screen for the current field
- Press the center mouse button (on three-button mice only) to simulate pressing <Enter>

You can also use the mouse when selecting from pick lists (see the section on TPPICK for details), when using the help system (see the section on TPHELP), and when responding to prompts for a single keypress (in general, clicking the right button does the same thing as pressing <Esc>, while clicking the left button does the same thing as pressing <Enter>).

M. Other Programs

Unlike most of the demonstration programs that come with Turbo Professional, which are intended to be useful as well as educational, two of them are pure fluff of the razzle-dazzle variety: WINEXMPL demonstrates some of the visual effects that can be created using the virtual screens managed by the TPSCREEN unit, and WINWOW illustrates the capabilities of the popup window manager in TPWINDOW, which also implements a set of routines for working with virtual screens.

WINEXMPL
WINEXMPL maintains five different screens (four of them virtual screens), and creates some interesting effects by merging the virtual screens with the physical screen in various ways. Small windows are popped onto the screen almost instantly, and the last one is then moved around the screen, giving the illusion that it is erasing the others. Press <Esc> to stop the wild window movement and return to DOS.

WINWOW
WINWOW starts by creating 8 popup windows of random sizes and colors, then begins moving them around the screen at a frenzied pace--scrolling, expanding, and contracting them as they go. Press any key to regain control of the helm and return to DOS.

14. Appendices

A. Unit Dependencies

The following is a list of all of the units in the Turbo Professional library and the lower level units that they depend on, if any. Units that may be overlaid using Turbo Pascal's overlay system are marked with an asterisk ('*').

TP8087
no dependencies

TPASCIIZ *
TPSTRING

TPBCD *
no dependencies

TPCRT
DPMI, TPINLINE

TPDATE *
DOS, TPINLINE

TPDIR *
DOS, TPCRT, TPDATE, TPDOS, TPINLINE, TPPICK, TPSTRING, TPWINDOW

TPDOS *
DOS, DPMI, TPSTRING

TPEARRAY *
DOS, TPEMS, TPMEMCHK

TPEDIT *
TPCMD, TPCRT, TPMOUSE, TPSTRING

TPEMS
no dependencies

TPENHKBD
DPMI

TPENTRY *
DPMI, TPBCD, TPCMD, TPCRT, TPDATE, TPINLINE, TPMEMCHK, TPMOUSE, TPSTRING, TPWINDOW

TPERRHAN
DOS

TPEXTMEM *
no dependencies

TPHELP *
DOS, TPCMD, TPCRT, TPDOS, TPMEMCHK, TPMOUSE, TPPICK, TPSTRING, TPWINDOW

TPINLINE
no dependencies

TPINT
DOS

TPINT24
no dependencies

TPMACED *
DOS, TPCRT, TPINT, TPMACRO

TPMACRO
DOS, TPINT

TPMEMO
TPCMD, TPCRT, TPMOUSE, TPSTRING

TPMENU *
TPCMD, TPCRT, TPMEMCHK, TPMOUSE, TPSTRING, TPWINDOW

TPMOUSE
DPMI, TPCRT

TPPICK *
TPCMD, TPCRT, TPMOUSE, TPSTRING, TPWINDOW

TPRARRAY *
TPINLINE, TPMEMCHK

TPSCREEN
DOS, DPMI, TPCRT

TPSORT *
no dependencies

TPSTRDEV
no dependencies

TPSTRING
no dependencies

TPTSR
DOS, TPINT

TPVARRAY *
DOS, TPMEMCHK

TPWINDOW *
TPCRT, TPINLINE, TPMEMCHK

Auxiliary units:

TPCMD
no dependencies

TPMEMCHK *
no dependencies

B. Keyboard Handling in Turbo Professional

Several units in Turbo Professional provide hooks that allow the programmer to control keyboard handling to some extent: TPEDIT, TPENTRY, TPPICK, TPMACED, TPMENU, TPHELP, and TPMEMO. More specifically, there are three kinds of hooks that fall into the category of keyboard handling.

The first is the *installable keyboard hook*. Six of the seven units (all but TPMACED) define a set of commands that are tied to certain keystrokes, and they all store the links between commands and keystrokes in an array that can be easily modified. The second is the *background task hook*. All seven units allow the programmer to specify a function that returns a keystroke, and that function is free to perform whatever tasks it wishes while waiting for a key to be pressed. The third is the *context-sensitive help hook*, which is provided by all of the listed units except TPHELP. These hooks facilitate the creation of context-sensitive help systems by providing a means of invoking a programmer-defined help routine when a particular command is issued.

For a good example of what can be done with these hooks, try compiling and running the demonstration program ENTRY.PAS. The installable keyboard hooks allow ENTRY to define new commands that can be executed while doing data entry (try moving the cursor to the "Zip" field and pressing <A>--you'll hear a beep--then press <AltB> and try it again. No more beeping). The background task hooks allow ENTRY to display an on-screen clock while input is being obtained from the keyboard. And the context-sensitive help hooks allow help screens to be popped up both while doing data entry and while choosing from a pick list of state names (try moving the cursor to the "State" field and pressing <F2> followed by <F1>). If you like what you see in ENTRY, then read on. Using these hooks is really quite easy.

Installable Keyboard Hooks

As indicated above, TPEDIT, TPENTRY, TPPICK, TPMENU, TPHELP, and TPMEMO all use typed constant arrays to store data that connects a particular key or sequence of keys with a command. Although these arrays are of different sizes, they are all of the same basic form as the EditKeySet array in TPEDIT:

```
EditKeyMax = 150;
EditKeyID : string[17] = 'tpedit5 key array';
EditKeySet : array[0..EditKeyMax] of Byte = (
 {length keys          command type      key sequence}
  2,     001,          RSwordLeft,       {^A}
  2,     004,          RSright,          {^D}
  3,     017, 004,     RSend,            {^Q^D}
  3,     017, 012,     RSrestore,        {^Q^L}
  3,     000, 071,     RShome,           {Home}
  3,     000, 075,     RSleft,           {Left}
```

EditKeyID is an ID string for use by installation programs; more on it later.

EditKeySet consists of a series of variably sized records, having three fields. The first is a number that indicates the combined size of the next two fields, the second contains the keycode(s), and the third contains the constant value associated with a particular command.

The keycodes field is the one that is variably sized. In most cases, commands are assigned either to a single key like <CtrlA> or <Home> or to a two-key combination like <CtrlQ><D>, and the code for a given key will occupy either one or two bytes. Control characters require one byte, while keys with extended scan codes (function keys, cursor keys, etc.) require two bytes: a 0 followed by the scan code. Hence the size of the keycodes field will usually be between 1 byte (a single control key) and 4 bytes (a combination of two keys with extended scan codes, like <AltD><AltR>). Although it is possible to assign combinations of three or more keys to a single command, doing so is inadvisable.

Now, given this information, there are three ways that you can customize the default command key assignments. The first and most obvious way is to simply edit the key array declaration for the Turbo Professional unit in question. This is also the best way if you want to make these changes permanent and to have them affect all your programs. If you do so, we strongly recommend that you change all six units (TPEDIT, TPENTRY, TPPICK, TPMENU, TPHELP, and TPMEMO) at once, and that you study the default key assignments for all of them before making any changes. The current key assignments maintain a high degree of consistency between units, a quality that you should try to retain when making your changes.

The second way to change them, and the one that is best for making minor changes for a particular program, is to use one of the following routines:

```
AddEditCommand  (TPEDIT)
AddEntryCommand (TPENTRY)
AddPickCommand  (TPPICK)
AddHelpCommand  (TPHELP)
AddMenuCommand  (TPMENU)
AddMemoCommand  (TPMEMO)
```

All of these routines are essentially the same as the AddEditCommand function in TPEDIT:

```
function AddEditCommand(Cmd : RStype; NumKeys : Byte;
                        Key1, Key2 : Word) : Boolean;
```

These routines all allow you to make three kinds of changes to the key assignments for a unit. First, they allow you to define new commands (which you have to process yourself, of course). Second, they allow you to disable a particular command. And third, they allow you to change a key assignment. We'll give you an example of each.

Let's say that you are using TPEDIT and want to allow your users to use <Up> and <Down> while editing to signal that the cursor should be moved to another edit field. (Presumably because you want to use TPEDIT to implement a simplified version of TPENTRY.) Here's how you would do it:

```
{assign first user-defined command to <Up>}
if not AddEditCommand(RSuser0, 1, $4800, $0000) then
  {error -- key array is already full} ;
{assign second user-defined command to <Down>}
if not AddEditCommand(RSuser1, 1, $5000, $0000) then
  {error -- key array is already full} ;
```

The first parameter is the command code. TPEDIT defines 30 commands (RSnone..RSuser9), and the last 10 (RSuser0..RSuser9) of them are set aside for user-defined commands. In this example, we're using the first two of them for <Up> and <Down>. The second parameter indicates whether the command is being assigned to a single key (such as <Up>) or a 2-key sequence (such as <CtrlQ><D>). The third and fourth parameters give the scan codes for the keys. In our example, both commands are being assigned to a single key, so the Key2 parameter will be ignored, and only the Key1 parameter is meaningful. If we were adding a 2-key WordStar-type command, say, <CtrlQ><D>, we would pass 'Ord(^Q)' (Key1) and 'Ord(^D)' (Key2). Notice that we would pass '^D' rather than 'D': TPEDIT will then accept either <CtrlD> or <D> following the <CtrlQ>, just as WordStar and all the Borland editors would.

Oh yes, to determine when <Up> or <Down> have been pressed, all you would need to do is to check the value of the global variable RSCommand after the call to ReadString (or ReadWord, or whatever Read routine you actually called). If RSCommand = RSuser0, <Up> was pressed, and if RSCommand = RSuser0, <Down> was pressed.

Now let's suppose that you wanted to disable a command, say, RSins (toggle insert mode). To do this you would reassign the <Ins> key, which is normally assigned to RSins, to 0, the null command (called 'RSnone' in TPEDIT):

```
{assign <Ins> to the null command}
if not AddEditCommand(0, 1, $5200, $0000) then
  {no error possible here} ;
```

Besides disabling the RSins command, this would also make more room available in the key array for adding user-defined commands. If you needed to reenable the command later, you could do so by executing the same statement, with RSins in place of 0 as the first parameter (Cmd).

Finally, let's say that you want to change an existing key assignment. For example,

maybe you want <CtrlHome> to do the same thing as <Home>, and <CtrlEnd> to do the same thing as <End>:

```
{assign <CtrlHome> to RShome}
if not AddEditCommand(RShome, 1, $7700, $0000) then
  {no error possible here} ;
{assign <CtrlEnd> to RSend}
if not AddEditCommand(RSend, 1, $7500, $0000) then
  {no error possible here} ;
```

When AddEditCommand scans the key array, it will see that <CtrlHome> is already assigned to RSdelBol (delete from beginning of line to cursor) and simply change its assignment to RShome. Similarly, <CtrlEnd> will be assigned to RSend rather than RSdelEol (delete from cursor to end of line). In both cases, the amount of space remaining in the array for expansion will be unchanged.

The third and final way to customize the key assignments is to write an installation program that allows your users to modify the key assignments to suit their fancy. The BONUS subdirectory contains a program, TPKEYS, that demonstrates how to do this. TPKEYS allows you to alter the key assignments for all five units with installable keyboard hooks. See the documentation for TPKEYS, also in the BONUS subdirectory, for further details.

Before moving on, we should point out that these same three methods can be used to change the way that these units respond to the pressing of a mouse button. If you look at TPEDIT.PAS, for example, you'll see the following near the end of the declaration for EditKeySet:

```
{$IFDEF UseMouse}
  3,     $00, $EF,    RSenter,    {click left  = Enter}
  3,     $00, $EE,    RSquit,     {click right = ESC}
  3,     $00, $ED,    RShelp,     {click both  = help}
{$ENDIF}
```

Thus, all you need in order to change the way the mouse buttons are interpreted are the pseudo-scan codes returned by the MouseKeyPressed routine in TPMOUSE:

```
MouseLft    = $EF00; {left button}
MouseRt     = $EE00; {right button}
MouseBoth   = $ED00; {both buttons}
{-------- three-button mice: ------}
MouseCtr    = $EC00; {center button}
MouseLftCtr = $EB00; {left and center buttons}
MouseRtCtr  = $EA00; {right and center buttons}
MouseThree  = $E900; {all three buttons}
```

You could, for example, change TPEDIT so that pressing all three mouse buttons on a three-button mouse does the same thing as pressing <CtrlR>:

```
if not AddEditCommand(RSrestore 1, MouseThree, $0000) then
  {error -- key array is already full} ;
```

Here again, you should give careful thought to the implications of the changes that you make to a given unit. The mouse button assignments in all six units are intended to provide, as much as possible, a consistent user interface: pressing the left mouse button makes a selection, pressing the right button is equivalent to pressing <Esc>, and pressing both buttons brings up a help screen (if there is one). We strongly urge you to try to maintain a similar level of consistency while making your changes.

Background Task Hooks

Many programmers like to have their programs perform background tasks (such as displaying the current time) while waiting for keyboard input. TPEDIT, TPENTRY, TPPICK, TPMACED, TPMENU, TPHELP, and TPMEMO all make this easy to do.

The first step is to write a function of the following form (if mouse support is being disabled):

```
uses TpCrt;
...
{$F+}
function GetKey : Word;
begin
  while not KeyPressed do begin
    inline($CD/$28);   {give TSRs a chance to pop up}
    {** perform background task here **}
  end;
  GetKey := ReadKeyWord;
end;
{$F-}
```

If you are using TPMOUSE in your program, the function would be slightly different:

```
uses TpCrt, TpMouse;
...
{$F+}
function GetKey : Word;
var
  I : Word;
begin
  I := $FFFF;
  repeat
    if KeyPressed then
      I := ReadKeyWord
    else if MousePressed then
      I := MouseKeyWord
    else begin
      inline($CD/$28); {give TSRs a chance to pop up}
      {** perform background task here **}
```

```
    end;
  until I <> $FFFF;
  GetKey := I;
end;
{$F-}
```

Notice that precedence is always given to pending keystrokes, rather than to mouse buttons. This insures that the pressing of a mouse button cannot interrupt a keyboard macro. (All of the units with background task hooks except TPMACED have mouse support.)

The second step is to tell the unit in question to call this routine when it needs to get a keystroke. In each case this is done simply by assigning the address of your routine to one of the following pointer variables:

```
EditKeyPtr     (TPEDIT)
EntryKeyPtr    (TPENTRY)
PickKeyPtr     (TPPICK)
MacEdKeyPtr    (TPMACED)
HelpKeyPtr     (TPHELP)
MenuGetKeyPtr  (TPMENU)
MemoKeyPtr     (TPMEMO)
```

For example, to instruct TPEDIT to use our GetKey routine:

```
EditKeyPtr := @GetKey;
```

Since each unit has its own hook, you can set up a different routine for each unit or use the same one for all of them. If you need to disable one or more of your routines at any point, you can do so simply by resetting the appropriate pointer variable to '@ReadKeyWord' (if mouse support is disabled) or '@ReadKeyOrButton' (if mouse support is enabled).

A few points about GetKey routines are worth noting. First, they *must* be declared FAR using {$F+}. Second, it is wise to call KeyPressed at the top of the polling loop rather than at the end; this insures that a background task won't get called if there is already a keystroke pending. Third, the "inline($CD/$28);" statement is sometimes necessary to allow TSR's to pop up while you're waiting for keyboard input. This statement generates an interrupt $28, which is a universal signal to TSRs that it's safe to pop up. We recommend that you always include this instruction in your GetKey routines so that you won't forget it in cases where it is needed. (TSRs are the most common cases.)

The ground rules for a background task are simple: the task must either execute rapidly, so as not to cause noticeable delays in the processing of keyboard input, or it must be interruptible. For example, let's say that your background task updates a series of windows, and that you have a procedure called UpdateWindow that

updates a specified window. The procedure that calls this routine might look like this:

```
procedure BackgroundTask;
  {-Update windows while waiting for keystrokes}
const
  WinNum : Byte = 1;
var
  I : Byte;
begin
  {$IFDEF UseMouse}
  while not KeyPressed or MousePressed do begin
  {$ELSE}
  while not KeyPressed do begin
  {$ENDIF}
    UpdateWindow(WinNum);
    if WinNum = WindowsToUpdate then
      WinNum := 1
    else
      Inc(WinNum);
  end;
end;
```

Before updating the next window, a check is made to see if a keystroke is pending (or a mouse button has been pressed, if appropriate). If so, BackgroundTask exits immediately to allow it to be processed. If not, it updates the window, then increments a counter that gets reset to 1 when all the windows have been updated. Because WinNum is a typed constant rather than a local variable, it retains its value between calls, so the next time BackgroundTask gets called it can pick up right where it left off. This is what we mean by "interruptible": the routine can give up control shortly after a key has been pressed, and it can resume execution of its task the next time that it is called.

One more point is worth making about GetKey routines. There's no law that says the value returned has to come from ReadKeyWord or MouseKeyWord. You might, for example, want to use this hook to implement a simple macro system. You could allow the user to define up to ten strings, stored in an array called MacroStrings, that are to be associated with the Alt-shifted function keys. If the user presses <AltF1>, you would return 'Ord(MacroStrings[1][1])' rather than <AltF1>; the next time GetKey was called, you would immediately return 'Ord(MacroStrings[1][2])'; and so on, until all the characters in MacroStrings[1] had been returned. Then you would go back to returning ReadKeyWord as usual.

Although not as flexible as the keyboard macro facility in TPMACRO, a simple macro system such as this can offer important benefits, especially in programs where users spend much of their time entering data into entry screens created with TPENTRY.

Context-Sensitive Help Hooks

To facilitate the use of TPHELP in your programs, TPEDIT, TPENTRY, TPPICK, TPMENU, TPMACED, and TPMEMO all define a help command (assigned to <F1> by default) and a hook that allows you to specify a routine that is to be called when that command is given. The steps involved in setting up this routine are similar to those required to gain control of keyboard input. Before we describe what those steps are, however, let us urge you to take time to first read the section on the TPHELP unit if you haven't already. The paragraphs that follow assume that you have at least a basic understanding of what that unit does and how to use it.

The first step is to write a routine that will process help commands. This routine should be a FAR-declared global procedure that takes three parameters:

```
{$F+}
procedure HelpRoutine(UnitCode : Byte; IdPtr : Pointer; HelpIndex : Word);
begin
  case UnitCode of
    HelpForPick,      {TPPICK}
    HelpForEdit,      {TPEDIT}
    HelpForMenu,      {TPMENU}
    HelpForEntry,     {TPENTRY}
    HelpForMacEd,     {TPMACED}
    HelpForMemo :     {TPMEMO}
     {adjust the HelpIndex as necessary, based on the UnitCode and IdPtr} ;
  end;

  {now process the help request}
  if not ShowHelp(MyHelpPtr, HelpIndex) then
    {handle the error} ;
end;
{$F-}
```

The first parameter, the unit code, will generally be equal to one of the constants shown in the example. These constants are defined in TPCRT (since that is the only unit that all five depend on). Here is the complete list:

```
HelpForUser  = 00;  {unused}
HelpForPick  = 01;  {TPPICK}
HelpForEdit  = 02;  {TPEDIT}
HelpForMenu  = 03;  {TPMENU}
HelpForEntry = 04;  {TPENTRY}
HelpForMacEd = 05;  {TPMACED}
HelpForMemo  = 06;  {TPMEMO}
HelpForXXXX2 = 07;  {reserved}
HelpForXXXX3 = 08;  {reserved}
HelpForXXXX4 = 09;  {reserved}
HelpForXXXX5 = 10;  {reserved}
```

When you need to call your help routine directly, you should pass either HelpForUser or a value greater than 10 as the UnitCode. The values from 6-10 are

reserved by Turbo Professional for future use. The purpose for the UnitCode will become clear momentarily.

The second parameter, the IdPtr, will usually be nil. In cases where the UnitCode is TPPICK or TPMENU, however, IdPtr will contain a meaningful address. When calling a help routine, TPPICK always passes the address of the routine that you (or perhaps TPDIR) specified as the StringFunc parameter to PickWindow. TPMENU passes the address of the current menu. In programs that contain multiple menus or multiple calls to PickWindow, these addresses can be used to determine what part of the program you were in when the request for help was made. TPMEMO passes the address of the current edit control block.

The third parameter, HelpIndex, also indicates what part of the program you are in. The source of the value varies widely from unit to unit, however:

- TPPICK: Passes the number of the item in the pick list that is currently highlighted.
- TPEDIT: Passes the current value of the global variable EditHelpIndex (it's up to you to initialize it before calling ReadString or one of the other Read routines).
- TPMENU: Passes the MenuKey for the currently highlighted item (a good reason for making the MenuKey for all items in all menus unique).
- TPENTRY: Passes the help index that you assigned to the field currently being edited (see Chapter 3, Section B).
- TPMACED: Always passes 0, on the assumption that the user is simply requesting help on the use of the macro editor; it is therefore up to your help routine to supply the correct index into the help file when UnitCode = HelpForMacEd.
- TPMEMO: Passes the help index that you specified when you called InitControlBlock.

Given the variety of meanings that might be attached to the IdPtr and HelpIndex parameters, it should now be clear what the purpose of the UnitCode is: to allow the help routine to determine, given the values in HelpIndex and IdPtr, what value should be passed to the ShowHelp routine as the Topic parameter.

That's the first step. The second step is to tell the unit in question to call this routine when the help command is issued. Again, this is done simply by assigning the address of your routine to one of the following pointer variables:

```
EditHelpPtr  (TPEDIT)
EntryHelpPtr (TPENTRY)
PickHelpPtr  (TPPICK)
MacEdHelpPtr (TPMACED)
MenuHelpPtr  (TPMENU)
MemoHelpPtr  (TPMEMO)
```

The default setting for each of these variables is nil, which in all cases but one causes the help commands to be ignored. In the case of TPMACED, pressing <F1> while in Command mode will insert that key into the macro being edited if MacEdHelpPtr is nil, or call the help routine it points to if it isn't. (In Literal mode the key will always be inserted, whether MacEdHelpPtr is nil or not.)

Once you've taken those two steps, all that remains is to write the text that will appear in your help screens and to run it through the MAKEHELP utility.

We realize, of course, that the preceding paragraphs contain a lot of information, and that you will probably need to read them more than once. Before you read them a second time, however, we strongly urge you to look at ENTRY.PAS and ENTRY.TXT (the ASCII file used by MAKEHELP to create ENTRY.HLP), and that you run ENTRY.EXE and experiment with its help system. Seeing how the help system is implemented in ENTRY should help you to understand the above documentation when you reread it.

C. Using Overlays with Turbo Professional

Unquestionably, one of the most significant enhancements in Turbo Pascal 5.0 and higher is the return of an overlay system. We know from our past experience with Turbo Extender, Overlay Manager 4.0, and other products that many Turbo Pascal programmers have large applications they need to maintain, and for them the use of overlays is less a matter of choice than a basic necessity. If you're one of the thousands who fall into this category, you should be pleased to hear that we've done everything we could think of to insure that Turbo Professional units will work well in overlayed programs.

The first and most important step we took in order to support the new overlay manager was to change or add to the compiler directives in all the units in the package. This meant that we had to set the {$F+} compiler directive in units that needed it (those that have to call user-written routines that might be overlayed) or {$F-} in those that didn't, and set the {$O-} directive in units that cannot be overlayed (usually because the unit contains an interrupt handler or a routine that might be called from one).

In many cases, however, we determined that a particular unit could be overlayed safely, and in these units we have specified the {$F+} and {$O+} directives. In alphabetical order, they are:

TPASCIIZ
TPBCD
TPDATE
TPDIR
TPDOS
TPEARRAY
TPEDIT
TPENTRY
TPEXTMEM
TPHELP
TPMACED
TPMEMCHK
TPMEMO
TPMENU
TPPICK
TPRARRAY
TPSORT
TPVARRAY
TPWINDOW

Before overlaying any of these units, you should be sure to study the list of unit dependencies given in Appendix A. In many cases, you will want to rule out certain units for performance reasons. For example, overlaying both TPWINDOW and TPMENU would be unwise, given that TPMENU needs frequent access to routines in TPWINDOW.

The second step we took was to modify the TPTSR unit to allow you to write memory resident programs that contain overlays. (We'll spare you the details of the internal changes that were required). It's not as difficult as you might think, as the brief program shown below (based on Borland's OVRDEMO.PAS) demonstrates:

```
{$V-,S-,I-,R-,F+,O+}
{$M 4096,0,650000}

program OvrDemo;

uses Overlay, Dos, TpDos, TpTsr, TpCrt, OvrDemo1, OvrDemo2;

{$O OvrDemo1}
{$O OvrDemo2}

const
  TabKey = $080F; {ALT-TAB}
  Count : Word = 0;
var
  OvrName : string[64];

  procedure PopProc(var Regs : Registers);
    {-This is the actual routine that pops up when ALT-TAB is struck.}
  begin
    if not ReleaseEms then
      if not ExistFile(OvrName) then
        {error--don't call any overlayed routines!} ;

    {increment counter}
    Inc(Count);
    WriteLn('Inside popup. Count = ', Count, '.');

    {call routines in overlayed units}
    OverlayRoutine1;
    OverlayRoutine2;

    {disable the program after popping up 10 times}
    if Count >= 10 then
      if DisableTsr then {success};
  end;

begin
  {locate overlay file}
  if not ExistOnPath('OVRDEMO.OVR', OvrName) then begin
    WriteLn('Cannot find OVRDEMO.OVR!');
    Halt(1);
  end;

  {open overlay file}
  OvrInit(OvrName);
  if OvrResult <> 0 then begin
    Writeln('Overlay error: ', OvrResult);
    Halt(1);
  end;
```

```
  {move overlay into EMS if possible}
  OvrInitEms;
  case OvrResult of
    ovrIOError: WriteLn('Overlay file I/O error.');
    ovrNoEMSDriver: WriteLn('EMS driver not installed.');
    ovrNoEMSMemory: WriteLn('Not enough EMS memory.');
  end;

  {if request for use of EMS succeeds, set ReleaseEms flag}
  ReleaseEms := (OvrResult = 0);

  {ALT Tab will activate this popup}
  if not DefinePop(TabKey, @PopProc, Ptr(SSeg, SPtr), True) then
    WriteLn('Unable to create popup routine')
  else begin
    WriteLn('Pop up loaded, type ALT-TAB to activate.');
    PopupsOn;
    if not TerminateAndStayResident(ParagraphsToKeep, 0) then
      WriteLn('Unable to go resident');
  end;
end.
```

Although this program is weak on error checking, it does demonstrate two important points. First, you must be absolutely certain that your program has access to its overlay file when you pop up. The best way to insure this is to obtain the full path name of the overlay file when your program starts, and use the ExistFile routine in TPDOS to check for its existence when you pop up. If the overlay file is not found, you should display an error message and exit from your popup--*without calling any overlayed routines*. Notice that checking for the overlay file is not necessary if EMS memory is being used to store the overlays.

Second, if you use the OvrInitEms routine to store the contents of your overlay file in EMS memory, you need to set the ReleaseEms variable in the TPTSR unit. This insures that any EMS memory allocated by the overlay manager will be released when you disable your TSR. Notice that it is usually not necessary to save and restore the mapping context of the Expanded Memory Manager, since Turbo 5's overlay manager handles that for you.

Except for the matter of error handling, which you will want to improve upon in your own TSRs, that about covers it. If you follow the two basic guidelines described above, you should have little difficulty writing a solid TSR that uses Turbo 5's overlay system.

Identifiers

A

AbsBCD 668
ActionCodeType 489
ActivateTiledWindow 88
ActivateVScreen 89
ActivateWindow 91
ActivePage 265
AddArrayField 324
AddBackSlash 424
AddBCD 668
AddBcdField 325
AddBooleanField 326
AddByteField 327
AddCharField 327
AddChoiceField 328
AddCompField 329
AddDateField 330
AddDateStField 331
AddDblField 332
AddExtField 332
AddFieldPrim 317
AddIntField 333
AddLongField 333
AddMemoCommand 414
AddNestedField 334
AddRealField 335
AddSglField 335
AddShortField 336
AddStringField 337
AddTimeField 338
AddUserField 339
AddWindowToLibrary 32
AddWordField 340
AddYesNoField 340
AllocateEmsPages 591
AllocateMacro 613
AllocateScreen 266
AllocateStack 531
AllowPendingSubmenus 117
AllowTruncation 410
AltPickAttr 238
ArcTanBCD 669
Asc2Str 466
AscCenter 466
AscCenterCh 467
AscCharStr 467
AscFromHeap 468
Asciiz 465
AsciizPtr 465
AscLeftPad 469
AscLeftPadCh 469
AscLoCase 470
AscPad 471
AscPadCh 471
AscToHeap 472
AscTrim 473
AscTrimLead 473
AscTrimTrail 474
AscUpCase 474
AssignCrt 25, 266
AssignMacroName 614
AuxKeyLength 612

B

BadDate 381
BadTime 381
BCD 667
BCDsize 667
BCDtoReal 669
BinaryB 424
BinaryL 425
BinaryW 425
BiosScroll 28
Black 24, 264
Blink 24
BlockCursor 32
BlockMouseCursor 212
Blue 24, 264
BMMakeTable 426

BMSearch 427
BMSearchUC 428
Brown 24, 264
BTable 423
BufLen 30
ButtonStatus 210
BW40 24
BW80 24

C

C040 24
C40 24
C80 24
CallOldIsr 693
CancelAllPrintFiles 490
CancelPrintFile 490
CanRecover 689
CaseChange 319
Center 429
CenterCh 429
CenterWrite 267
ChainInt 532
ChangeAttribute 33
ChangeAttributeWindow 33
ChangeDirectory 256
ChangeFieldAttr 341
ChangeHotKey 554
ChangePromptAttr 341
ChangeProtection 342
ChangeProtectionFast 343
ChangeRequired 343
ChangeStringAttr 344
ChangeValidation 344
CharSet 159, 319, 423
CharStr 430
CharToMacro 614
CheckBreak 25
CheckEOF 25
CheckKbd 34
CheckMenuStatus 121
CheckSnow 25
CleanPathName 430
ClearA 633, 645, 654
ClearEntryWindow 345
ClearFirstChar 157
ClearFlag 693
ClearMacros 615
ClearVScreen 91
ClearWindow 116
CloseHelp 182
CloseLibrary 34
ClrEol 25, 262
ClrScr 25, 262
CO80 24
ColonChar 381
ColorColors 267
CommaChar 320
CommaForPeriod 423, 667
CommandSet 319
CompareLetterSets 431
CompareType 423
CompAsc 475
CompString 433
CompStruct 434
CompUCAsc 475
CompUCString 435
ConcatAsc 476
ConcatStr 476
CopyAsc 477
CopyScreen 268
CopyVirtualScreenToWindow 317
CopyWindow 269
CosBCD 670
CreateLibrary 35
CrtError 28
CtrlBreakFlag 30
CurrencyLtStr 320
CurrencyRtStr 320
CurrentCols 271
CurrentDisplay 30
CurrentFieldModified 345
CurrentMode 30
CurrentPage 30

CurrentRows 271
CurrentScreen 265
CurrentTime 385
CurrentTimeString 385
CurrentWindow 87
CursorEndLine 36
CursorStartLine 36
CursorToEnd 157
CursorTypeSL 37
Cyan 24, 264

D

DarkGray 24, 264
Date 381
Date1900 382
Date1980 382
Date2000 382
DateDiff 386
DatePicture 254
DateString 381
DateStringToDate 386
DateStringToDMY 387
DateTimeDiff 388
DateTimeRec 381
DateToDateString 389
DateToDMY 389
DateToSortString 390
DayOfWeek 390
DayString 382
DayType 381
DeactivateVScreen 92
DeallocateEmsHandle 592
DeallocateMacro 615
DeallocateScreen 272
DeallocateStack 533
DecimalChar 320
DefaultDrive 491
DefaultExtension 435
DefaultMonth 384
DefaultYear 384
DefinedKeys 612
DefineMacro 616
DefinePop 555
DefinePopProc 556
DefineScreen 272
Delay 25
DeleteAsc 477
DeleteJoinsLines 410
DeletePop 557
DeletePopProc 557
DeleteWindowFromLibrary 38
DelLine 25
deName 27
DeTab 436
DetectMultitasking 29
DirDisplayStr 254
DirectoryEntry 27
DirectoryPtr 27
DirectoryType 27
DirectVideo 25
DirsUpper 254
DisableEditMouse 160
DisableEntryMouse 346
DisableEventHandling 212
DisableHelpIndex 182
DisableHelpMouse 183
DisableMemoMouse 414
DisableMenuItem 122
DisableMenuMouse 124
DisablePickMouse 242
DisableTSR 558
DisallowedInReadOnlyMode 410
DiskClass 489
DisplayOverride 29
DisplayType 27
DisplayWindow 93
DisposeA 633, 645, 654
DisposeAsc 478
DisposeEditScreen 346
DisposeMenu 124
DisposePackedWindow 40
DisposeString 437
DisposeSubMenu 116

DisposeVScreen 94
DisposeWindow 94
DispPackedWindow 39
DispPackedWindowAt 39
DivBCD 670
DMYtoDate 391
DMYtoDateString 391
DosBusyFlag 559
DosCriticalFlag 559
DOSVersion 491
DrawEditScreen 347
DrawField 347
DrawFrame 116
DrawItem 116
DrawLine 413
DrawMenuFromStack 125
DrawShadowFrame 85
DrawSubMenu 116
Dummy5 530

E

EditButtonCodes 159
EditHelpIndex 159
EditHelpPtr 159
EditKeyPtr 159
EditKeys 627
EditMapCtrls 157
EditMemo 415
EditScreen 348
EditSize 157
EditString 349
ElementsSorted 582
EMcontrolBlock 412
EmsActiveHandles 593
EmsAvailable 644
EmsErrorCode 590
EmsInstalled 593
EmsPageFramePtr 594
EmsPagesAvail 595
EmsPagesOwned 596
EmsStatusOK 597
EmsTotalPages 598
EmsVersion 598
EMtype 412
EmulateInt 534
EnableEditMouse 160
EnableEnhanced 522
EnableEntryMouse 351
EnableEventHandling 213
EnableHelpIndex 183
EnableHelpMouse 184
EnableMemoMouse 417
EnableMenuItem 125
EnableMenuMouse 126
EnablePickMouse 243
EndOfMacro 611
EndPrintStatus 492
EnhancedDisplay 30
EnTab 438
EntryHelpPtr 320
EntryKeyPtr 323
EntryMapCtrls 320
EqualBCD 671
EraseCurrentSubMenu 128
EraseMenu 129
EraseMenuOntoStack 130
EraseSubMenu 116
EraseTopWindow 95
Error8087 570
ErrorA 634, 646, 655
ErrorAttr 410
ErrorRecovery 690
ErrorRow 410
ESrecord 319
EStype 319
EvaluateEScommand 352
Exceptions8087 571
ExchangeBytes 694
ExchangeStructs 694
ExchangeWords 695
ExecDos 493
ExistFile 495
ExistOnPath 496

ExpBCD 672
Explode 86
ExplodeDelay 86
ExplodeFrame 85
ExtMemAvail 603
ExtMemTotal 603
ExtractWord 439

F

FarCall 695
FastCenter 40
FastFill 41
FastFillWindow 41
FastFlush 42
FastRead 43
FastReadLine 273
FastReadWindow 43
FastText 44
FastTextWindow 44
FastVert 45
FastVertWindow 45
FastWrite 46
FastWriteClip 96
FastWriteCtrl 413
FastWriteMem 273
FastWriteStack 97
FastWriteWindow 47
FatCursor 47
FatMouseCursor 213
FieldRec 319
FileHandlesLeft 497
FileHandlesOpen 497
FilesUpper 254
FillAttribChar 274
FillPickWindow 245
FillStruct 696
FillVideoChar 274
FillWindow 275
FillWord 696
FiltersEnhanced 522
FindFieldID 317
FindHelp 186
FindLineIndex 413
FindLineLength 413
FindMacroIndex 616
FindMenuItem 131
FlagIsSet 697
FlexAttrs 27
FlexWrite 48
FlexWriteWindow 49
Float 423
FlushA 657
Font8x8 24
Font8x8Selected 49
ForceCaseUser 321
ForceExtension 439
ForceUpper 157
Form 440, 672
FracBCD 673
FrameArray 27
FrameChars 29
FrameCharType 27
FrameDelta 117
FrameWindow 50, 263
FreeExtMem 604
FullMouseWindow 214
FullPathName 442

G

GetCrtMode 50
GetCurrentMenuKey 131
GetCursorState 51
GetDiskClass 498
GetDiskInfo 499
GetDta 500
GetElement 583
GetEnvironmentString 501
GetExtMem 604
GetFileMode 502
GetFileName 257
GetLine 413
GetMenuItemName 132

GetMickeyCount 214
GetMousePage 215
GetPrintStatus 503
GoToXY 25, 262
GotoXYAbs 51, 263
GreaterBCD 673
GreaterEqualBCD 674
Green 24, 264

H

HaltOnError 632, 644, 653
HandleIsConsole 504
HardMouseCursor 216
HasEnhancedKbd 522
HelpCmdNum 181
HelpColorArray 179
HelpColorType 179
HelpForMemo 410
HelpKeyPtr 181
HelpMore 180
HelpMouseScroll 181
HelpPtr 179
HelpTitle 180
HercCardType 28
HercGraphicsMode 52
HercModeTestWorks 52
HercPresent 53
HexB 443
HexL 443
HexPtr 444
HexW 444
HiddenCursor 54
HiddenMouseCursor 217
HideCursor 117, 181, 238
HideCursorInReadChar 157
HideMouse 217
HighVideo 25, 262
HiWord 697
HKtype 179
HMStoTime 392
HoursInDay 383
HouseCursorAtEnd 157
HugePtr 602

I

IfcPtr 552
IfcRecord 552
ImplodeFrame 85
IncDate 392
IncDateTime 393
IncTime 393
IndentStartsParagraph 410
InitBufferState 413
InitControlBlock 417
InitESrecord 353
InitializeMouse 218
InitKbdVectors 523
InitMacros 617
InitPictureFlags 317
InitPops 560
InitType 632, 644, 653
InitVector 535
InitVideo 275
InsertAsc 479
InsertByDefault 157
InsertMode 321
InsertStr 479
InsLine 25
Installed8087 572
InstallModule 560
Int24Off 576
Int24On 576
Int24Result 561, 577
IntBCD 674
InternationalCurrency 354
InternationalDate 394
InternationalTime 395
InterruptsOff 536
InterruptsOn 536
InTextMode 30
IntnnHandle 552
IntRegisters 530

IntReturn 537
IoResultPtr 489
ISR_Array 530
ISR_Record 530

J

JumpRecord 692
JumpToOldIsr 698
JustExtension 445
JustFilename 445
JustPathname 446

K

KbdFlags 54
KeepDiskCurrent 653
KeyPressed 25
KeyToString 617
KillTiledWindow 98
KillWindow 98

L

LastEntryCommand 323
LastMode 25
LeftPad 447
LeftPadCh 447
LenAsc 480
LessBCD 675
LessEqualBCD 675
LetterValues 423
LibName 27
LightBlue 24, 264
LightCyan 24, 264
LightGray 24, 264
LightGreen 24, 264
LightMagenta 24, 264
LightRed 24, 264
LnBCD 676
LoadA 636, 648, 658
LoadVA 637
LoCase 448
LoCaseMac 448
Long2Str 449
LongintToBCD 676
LongJump 699
LotusFrame 117
LoWord 700
LowVideo 25, 262

M

MacEdKeypressedPtr 626
MacroAddress 618
MacroCount 618
MacroFileID 611
MacroInProgress 612
MacroNames 612
MacroPointers 612
MacroRec 611
MacroRecording 612
MacroRecordingOff 619
MacroRecordingOn 619
MacroRecPtr 611
MacrosAreOn 612
MacrosOff 620
MacrosOn 620
Magenta 24, 264
MakeA 638, 649, 659
MakeAPrimitive 639, 661
MakeLetterSet 449
MakeVScreen 99
MakeWindow 100
MapColor 55
MapColors 29
MapEmsPage 599
MapPackedWindowColors 56
MarkExtMem 605
MaxAsciiz 465
MaxDate 383
MaxISRs 530
MaxKeysInMacro 611
MaxLineLength 410

MaxMacros 611
MaxPopups 552
MaxScreens 264
MaxSearchLen 238
MaxTime 383
MaxWord 700
MaxYear 383
MemoError 418
MemoErrorPtr 410
MemoHelpPtr 411
MemoKeyPtr 411
MemoKeySet 411
MemoMouseEnabled 411
MemoStatus 418
MemoStatusPtr 411
MemoStatusType 412
Menu 119
MenuCharSet 119
MenuChoice 133
MenuCmdNum 120
MenuColorArray 119
MenuColorType 119
MenuGetKeyPtr 120
MenuHeight 135
MenuHelpPtr 120
MenuItem 136
MenuItemProc 117
MenuItemPtr 137
MenuKey 119
MenuMode 138
MenuStatus 139
MenuWidth 139
MergePicture 317
MinDate 383
MinSpaceForDos 489
MinTime 383
MinutesInHour 383
MinWord 701
MinYear 383
MKType 120
ModuleInstalled 561
ModulePtrByName 562
MoneySign 423, 667
Mono 24
MonoColors 276
MonthString 383
MouseButtonPressed 219
MouseButtonReleased 219
MouseCursorOn 211
MouseEventType 210
MouseGotoXY 220
MouseInstalled 211
MouseInWindow 220
MouseKeyWord 221
MouseKeyWordX 211
MousePressed 222
MouseStateBufferSize 222
MouseStatePtr 210
MouseWhereX 223
MouseWhereXY 223
MouseWhereY 224
MouseWindow 224
MouseXLo 210
MoveExtMem 606
MoveScreen 57
MoveVScreenToWindow 102
MoveWindow 104
MoveWindowHorizontal 277
MoveWindowToVScreen 105
MoveWindowVertical 278
MultBCD 677
MultiTaskingOn 30

N

NearCall 701
NewMenu 140
NoFrame 117
NoHelp 118
NonMacroKey 626
NormalAttr 30
NormalCursor 58
Normalized 702
NormalMouseCursor 225

IDENTIFIERS

NormVideo 25, 262
NoSound 25
NotEqualBCD 677
NotFound 465
NumberOfDrives 504

O

OctalB 450
OctalL 450
OctalW 451
OneMS 31
OpenHelpFile 187
OpenHelpMem 189
OpenLibrary 58
OpenStdDev 505
Orientation 120
OutputToScreen 265

P

PackedScreen 27
PackedWindow 28
PackedWindowPtr 28
PackLibrary 59
PackMenuLib 142
PackWindow 60
Pad 451
PadCh 452
ParagraphsToKeep 562
ParsePath 505
PasswordChar 157
PasswordMode 158
PiBcd 667
PickAlterPageRow 238
PickBar 247
PickChar 241
PickCmdNum 241
PickColorArray 237
PickColorType 237
PickHelp 191
PickHelpPtr 241
PickKeyPtr 241
PickMatrix 238
PickMaxRows 239
PickMinRows 239
PickMore 239
PickMousePage 239
PickMouseScroll 239
PickMouseWindow 239
PickOrientation 247
PickSrch 240
PickSrchStat 240
PickStick 240
PickUserPtr 241
PickWindow 248
PKType 237
PlaybackDelay 611
PopSubLevel 143
PopSubMenu 116
PopTickerPtr 552
PopupInUse 552
PopupsEnabledPtr 553
PopupsOff 563
PopupsOn 563
PopupToCallPtr 553
PosAsc 480
PosStr 481
PrintInstalled 506
PtrToHuge 607
PushSubMenu 116
PutElement 584

R

RangeCheck 632, 644, 653
ReadAttrAtCursor 61
ReadAttribute 62
ReadAttributeWindow 62
ReadCharacter 161
ReadCharAtCursor 63
ReadDiskSectors 507
ReadInteger 162
ReadKey 25

ReadKeyOrButton 225
ReadKeyWord 64
ReadLnAsc 482
ReadLongInt 163
ReadMacroFile 621
ReadMemoFile 419
ReadMenuLib 144
ReadPackedWindow 65
ReadReal 164
ReadStr 485
ReadString 165
ReadVScreen 106
ReadWindowFromLibrary 66
ReadWord 166
Real2Str 452
RealToBCD 678
Red 24, 264
RedrawMenu 145
RegKeyLength 612
Reinit8087 572
ReinitCrt 67
ReinitEmulator 573
ReinitScreen 279
ReleaseExtMem 607
ReleaseVideo 280
RemoveMacros 621
RemovePops 564
ResetEntryScreenFlags 355
ResetMenu 146
ResizeWindow 107
Restore8087 573
RestoreAllVectors 538
RestoreCursorState 67
RestoreEmsContext 600
RestoreKbdVectors 523
RestoreMouseState 226
RestoreShadowFrame 85
RestoreVector 538
RestoreWindow 68
RestoreWindowCoordinates 68, 281
RetA 640, 650, 662
ReturnCompletePath 255
ReturnStr 485
RingBell 355
RoundBCD 679
RoundToNearestHour 395
RoundToNearestMinute 396
RSChWord 159
RSCommand 159
RSpadChar 158
RStype 159

S

Save8087 574
SaveBuffer8087 569
SaveEmsContext 601
SaveInt1B 25, 31
SaveMemoFile 419
SaveMouseState 226
SaveShadowFrame 85
SaveWindow 69
ScrapMacro 612
ScrapMacroKey 612
ScrapMacroName 612
ScreenHeight 31
ScreenHelp 192
ScreenRec 264
Screens 265
ScreenWidth 31
ScrollByPage 323
ScrollWindow 108
ScrollWindowDown 70, 263
ScrollWindowUp 71, 263
Search 453
SearchUC 454
SecondsInDay 383
SecondsInHour 383
SecondsInMinute 383
SelectDrive 509
SelectFont8x8 72
SelectMenuItem 146
SelectOnFirstMouseClick 118
SelectScreen 281

IDENTIFIERS

SendEOI 539
SeparateDirs 254
SetA 641, 651, 663
SetActivePage 282
SetAllFieldLinks 356
SetAutoAdvance 358
SetAutoNumLock 359
SetBackgroundChar 282
SetBeepOnError 359
SetBell 360
SetBlink 73
SetBlock 509
SetClearFirstChar 360
SetCrtBorder 73
SetCtrlAttr 361
SetCursorSize 74
SetCursorToEnd 361
SetDelimiters 362
SetDta 510
SetEntryWindow 362
SetErrorPtr 363
SetExitOnSecondClick 364
SetFieldAttr 364
SetFieldLinks 365
SetFlag 702
SetForceMode 366
SetFrameChars 74
SetHelpPos 194
SetHercMode 75
SetHorizontalPick 250
SetInactiveFrame 108
SetInsertPushes 366
SetJump 703
SetMenuDelay 147
SetMenuItemName 148
SetMenuMode 149
SetMenuSelectKeys 150
SetMickeyToPixelRatio 227
SetMouseEventHandler 228
SetMousePage 229
SetNumeric 367
SetPadChar 367
SetPasswordChar 368
SetPasswordMode 368
SetPopTicker 544, 564
SetPostEditPtr 369
SetPreEditPtr 370
SetPromptAttr 370
SetProtectAttrs 371
SetProtection 371
SetRawMode 511
SetRequired 372
SetRightJustify 372
SetSnakingPick 250
SetStringAttr 373
SetTopTiledWindow 109
SetTopWindow 110
SetTrimBlanks 373
SetUpdatePtr 374
SetVerify 512
SetVerticalPick 251
SetVisiblePage 75
SetWaitCount 622
SetWrapMode 375
Shadow 86
ShadowAttr 87
ShadowMode 87
ShadowType 86
ShowExtension 254
ShowHelp 195
ShowMouse 229
ShowPrevHelp 198
ShowReadChar 158
ShowSizeDateTime 254
SideKickLoaded 552
SinBCD 680
SizeDisplay 254
SlashChar 383
SoftMouseCursor 230
Sort 585
SortStatus 582
SortStringToDate 396
SortStringToTime 397
Sound 25

Soundex 455
SoundFlagW 87
SpeedWrite 283
SqrBCD 681
SqrtBCD 681
SrchStart 240
SrchString 237
SrchType 237
StackSafetyMargin 489
StartMacro 622
StartMacroByKey 623
StatusAttr 411
StatusRow 411
StdErrHandle 489
StdInHandle 489
StdOutHandle 489
StdPrnHandle 489
StLoCase 455
StoreA 642, 652, 664
StoreVA 643
StoreWindowCoordinates 76, 284
Str2Asc 483
Str2Int 456
Str2Long 456
Str2Real 457
Str2Word 457
StrBCD 682
StrExpBCD 683
StringFromHeap 458
StringPointer 611
StringToHeap 459
StringToMacro 624
StringToScrapMacro 624
StripPicture 317
StuffKey 77
StuffString 77
StUpCase 460
SubBCD 683
SubMenu 151
SubmitPrintFile 513
SubstChars 323
SwapNibble 704
SwapStackAndCall 540
SwapStackAndCallNear 541
SwapWord 704
SwitchInColorCard 78

T

TempMacro 626
TerminateAndStayResident 565
TextAttr 25
TextBackground 25, 262
TextChar 30
TextColor 25, 262
TextFileSize 514
TextFlush 514
TextMode 25, 262
TextPos 515
TextSeek 515
ThisIfc 553
Threshold2000 384
Time 381
TimeDiff 397
TimeMs 516
TimePicture 254
TimeStringToTime 398
TimeToAmPmString 398
TimeToHMS 399
TimeToSortString 399
TimeToTimeString 400
Today 400
TodayString 401
TpArray 632, 644, 653
TpEntryScrolls 309
TPStr 484
Trim 460
TrimBlanks 158
TrimLead 461
TrimTrail 461
TruncBCD 684
Turbo3StyleColors 264

U

UninstallModule 565
UpCase 462
UpCaseMac 462
UpCaseTime 384
UseCrt 23
UseEmulator 574
UseFileFrame 255
UseHelpFrame 180

V

ValBCD 685
ValidateNoBlanks 376
ValidateNotPartial 376
ValidateSubfields 377
ValidDate 401
VerifyOn 516
VideoSegment 31
VirtualSegment 31
VirtualWidth 31
VisiblePage 265
VScreen 86
VScreenCols 111
VScreenRows 111

W

WaitForButtonRelease 211
WhereX 26, 262
WhereXAbs 79, 263
WhereXY 79, 263
WhereXYdirect 80
WhereY 26, 262
WhereYAbs 81, 263
WhichHerc 31
White 24, 264
WindMax 25
WindMin 25
Window 26, 262
WindowCoordinates 28, 264
WindowIsActive 112
WindowPtr 86
WindowRelative 158
WordCount 463
WordWrap 464
WrapModeType 320
WriteAsc 483
WriteAttribute 81
WriteAttributeWindow 82
WriteDiskSectors 517
WriteError 632, 644, 653
WriteMacroFile 625
WriteMenuLib 152
WritePackedWindow 82
WriteVScreen 113

Y

Yellow 24, 264
YesOrNo 167

Z

ZeroBcd 667

IDENTIFIERS

Index

0

$$WORK$$.LBR 59
<CtrlBreak> 30, 545, 551
<NumLock> 710
{$F+} 6, 690, 695, 701, 763
{$I-} 546, 575, 688
{$M} 546, 688
{$N+} 329, 332, 335
{$O+} 763
6845 CRT controller 80
80287 526
80387 526
8087 526, 566
8x8 font 30, 49, 72

A

Absolute screen coordinates 22
Action codes 493
Active display page 260
Active page 263, 282
Active window 84
Alarm 715
Alternate stack 525, 529, 531, 540, 544
ASCIIZ 421, 465
Assembly language routines 3
Assumptions 6
Attributes, file 502
Attributes, video 33, 46, 55, 84, 721, 722
Auto-detection of display adapters 29
AutoAdvance 358
AutoNumLock 359

B

Background character 262, 282
Background task hook 753
Background tasks 311, 745, 757
BCD 325, 665, 710
Benchmark 630
Bickel, Michael 431
Binary Coded Decimal 665, 710
Binary Editor 308
BINOBJ.EXE 177
BiosScroll 70, 71
Blank trimming 373
BLDGUIDE.EXE 5
Blinking text 73, 722
BONUS 5
Boyer-Moore algorithm 426, 427
BSORT 735
BufLen 21
Burns, Philip R. 666

C

Calculated Fields 306
CGA emulation 78
ClearFirstChar 360
Color mapping 29, 84, 260
Command entry pointer 534, 549, 562
Command, last 304
Compaq 21
Compiler support 9
Compression 60
CompuServe 18
COMSPEC 487
Conditional compilation directives 6
CONFIG.SYS 497
Context 587, 600, 601
Context-sensitive help 168, 311, 745, 753, 760
Control characters 154, 302, 361
Conversion routines 288, 313
Copying B-Tree Filer for C 17
Critical error 527, 546, 551, 561, 575
CRT, compatibility of TPCRT with 22
CRT unit 19, 262
Currency 298, 320, 354
Current time 385

Current window 84
Cursor 51, 58, 67, 72, 84, 100
Cursor, hidden 54
Cursor mask 201
Cursor position 80
Cursor scan lines 32

D

Data entry flow 287
Data entry screens 745
Data entry screens, nested 307, 334
Data entry window 308, 362
Data types 285
Database Toolbox 390, 399, 579
Date arithmetic 378, 381
Davis, Dan T. 234
Day of the week 382, 390
DBase 295
Delimiters 362
DesqView 19, 22, 29
Dettmann, Terry 7
Devices 505
DIFF 741
Directories 510
Directory, change 256
Directory name 424
Directory window 252
DirectVideo 24
Dirty pages 657
Disabling a TSR 558
Disabling menu items 122
Disk class 498
Disk sectors 507, 517
Disk Transfer Address 510, 551
Display adapters 27, 30
Display pages 19, 21, 30, 75, 215, 229, 260, 265, 282
DOS busy flag 559
DOS critical flag 559
DOS services 542, 545, 550, 555
DOS shell 493, 571
DOS unit 487
Double DOS 29
DTA 510, 551
Duncan, Ray 7

E

Edit buffer 402
Edit window 402
Editing commands 153, 155, 165, 300
Editor control block 402
Editor Toolbox 308
EGA 30, 36, 47, 49, 72, 74, 75
EKFIX.COM 520
EMM 587
EMS 548, 587, 629, 630, 765
EMS page frame 594, 599
EMS pages 587, 591, 639
Emulator 688
End Of Interrupt 539
Enhanced keyboards 488, 518
Enumerated types 307
Environment variables 501
EOI 539
Error Handling 310
Event handler 205, 213, 228, 243
Exception interrupts 568, 570, 571
ExecDos 67, 279, 487, 571
Exit commands 118, 158, 240, 289, 294, 300
Exit procedures 6, 280, 592, 688, 690
ExitProc 6
Expanded memory 587, 741
Expanded Memory Manager 587
Expanded Memory Specification 587
Exploding window 86
Extended memory 587, 606, 741
Extension 435, 439, 445

F

FAR model 6, 493, 586
FASTUPD.nnn 4

Field attribute, change 341
Field delimiters 362
Field editors 300
Field ID 293
Field movement commands 300
File pointer 515
FileMode 495
Files supplied 4
Filter 504
Flicker 22, 28, 57, 70, 71
Floating point emulation 527, 548, 566
Floating point overflow 666
Formatted strings 422, 440, 484
Formatting characters 294, 379
Frame characters 50, 74, 100, 362
Free list 488

G

GRAPH unit 23
GRAPH3 22
Graphics 21, 24, 30, 63, 67
GUIDE.DOC 5

H

Handles, EMS 591
Handles, file 489, 497
Handles, ISR and popup 525, 528, 535
Handles, screen 260
Hard disk 3
Heap overflow 688
Heap size 546
HeapError 688
Help file directives 170
Hercules 27, 30
Hercules graphics card 28
Hercules graphics mode 52, 75
Hercules InColor card 78
Hexadecimal arithmetic 543
Hidden field 315
Hotkeys 115, 543, 554, 708, 726
Huge pointers 588, 607

I

Implementation 6
Inline macros 6, 609, 687, 692
INSTALL 4, 11
Installable keyboard hook 753
Installation program 756
Interface 6
Interrupt procedures 6, 525, 528, 693, 698
Interrupt Service Routine 525, 528, 535
Interrupt vectors 6, 535
Interruptible 759
ISR 525, 528, 535
ISR handles reserved 528

J

Jourdain, Robert 7
Julian date 380
Jump table 695, 701

K

Key Assignments 314
Key Codes 518
Keyboard buffer 77
Keyboard handling 753
Keyboard macros 609, 610
Keyboard status flags 54

L

LBR file format 35
Letter set 431
LHARC.EXE 4
Libraries 35
Line editor 153
Linear addresses 588, 602, 607
Links, field 356
Literal characters 294

Logical pages 594, 599
Lotus-style menus 117

M

Macro Assembler 3, 667
Macro editor 627
Macro engine 617
Macro file 621, 706
Macro name 614
Macro recording 619
Macros, keyboard 609, 610
Macros, nested 609
MAKE 4, 11
Make file 10
MAKEHELP 169, 175, 762
MAKEMENU 115, 136, 142, 201, 724
Map colors 29, 84, 260
Mapping context 587, 600, 601
MASM 3, 667
MCGA 30, 75
Memo fields 402
Memory management 581
Memory requirements 3
Memory resident programs 6
Menu ID 144
Menu item 115
Menu item, disabling 122
Menu library 142, 144, 152, 727, 733
Menu selection keys 150
Mickey 209, 214
Mickey-to-pixel ratio 209, 227
Microsoft Windows 29
Mistakes, common 581, 631
Modules 548, 560
Monitor 3
Mouse buttons 202, 221, 225
Mouse cursor 200, 216, 230
Mouse drivers 208, 723
Mouse droppings 204
Mouse event handler 205, 213, 228, 243
Mouse support 115, 126, 154, 184, 195, 200, 231, 309, 311, 351, 720, 743
Mouse window 203, 214, 224
Multiple-choice editor 300
Multiple-choice fields 306, 328
Multitasking 29

N

NEAR 6
Network 495
Normalized pointer 702
Norton, Peter 6
Numeric coprocessor 526, 688
Numeric editor 300, 367
Numeric keypad 64

O

OmniView 19, 22, 29
On-screen clock 716
OPSTACK 548
OS/2 551
Overflow errors 568
Overlay errors 688
Overlay Manager 763
Overlays 548, 751, 763

P

Page frame 594, 599
Password mode 368
PATH 442, 487, 496
PCALC 710
Periscope 546
Phillips, Carley 22
Physical pages 594, 599
PIC 539
Pick list orientation 234, 247
Pick lists 231, 745
PickHorizontal 234
PickSnaking 234

PickVertical 234
Picture mask 254, 294, 378
Playback delay 708
Pop ticker 564
Popup dispatcher 542
Popup handles 543
Popup routines 542
Popup window 50, 749
Post-edit routine 289
Pre-edit routine 288
PREF 201, 721
PRINT.COM 487, 506, 513, 709
PRINTDOC.EXE 4
Printer errors 577
Program Segment Prefix 551
Programmable Interrupt Controller 539
Prompt attribute, change 341
Protected fields 235, 306, 342, 371
Protected mode 9, 60, 587
Protection 343
PS/2 3
Pseudo-scan codes 203, 221, 225, 721, 756
PSP 551
PTIME 715
PTRPT 718
Purchase Agreement 17

Q

Quicksort 579

R

RAMDRIVE.SYS 588, 589
Random access 487
Read-only mode 322
READ.ME (DEMO) 4, 13
READ.ME! (BONUS) 5
Redefinable keyboard 231
Redirection 738
Reentrancy 206, 408, 529, 550
Registers 544
ReinitCrt 279
ReleaseEms 765
REP 738
Required field 305, 372
Required status 343
Restore a vector 538
Right justification 372
Run-length encoding 60
Runtime errors 546, 568, 687, 688, 690

S

Scan code 64, 543
Scan lines 37, 74
Screen coordinates 46
Screen mask 201
Scroll bar 181, 195, 231, 239, 309, 722
Scrollable entry screens 309
Scrolling 28, 54, 70, 71, 308
Segmented addresses 588, 607
Select menu items 140
Semi-literal characters 294, 295, 379
Shadowed window 87
Shift keys 543
SideKick 520, 550
SideKickPlus 550
Slider 239
SMACS 706, 715
Snow 22, 28, 57, 70, 71
Sortable string 390, 396, 399
Sorting 579, 694, 735
Soundex 421, 431, 455
Stack and heap usage 547
Stack overflow 688
Stacked windows 83, 84
Stacks 525, 529, 531, 540, 544
Standard input 489, 505
Standard interface 548, 560
Standard output 489, 505
Status word 568, 570
Strenczewilk, Don 427
String attribute, change 344

String comparison 433
String editor 300
String manipulation 421
Subfield 299
Submenu 115
Support, telephone or CompuServe 18
System requirements 3

T

TaskView 29
TASM 3
TDebugPLUS 546
Telephone support 18
Terminate and stay resident 6, 525, 542, 565
Text editor 402
Text file device driver 484
Text files 487
TFIND 736
Tiled windows 84
TopView 29
TPC.CFG file 11
TPKEYS 756
TPRO.MAK 4, 11
TPTIMER 516
TPU files 9
TSR 6, 21, 67, 207, 279, 493, 525, 542, 565, 587, 758
Turbo Assembler 3, 667
Turbo Debugger 546
Turbo Extender 763
TURBO3 22
Turbo3StyleColors 262
TURBO5.MAC 706

U

Unhidden 54
Unit dependencies 751
Unit initialization blocks 6
Unit name override 629
Units 6
Unloading a TSR 549, 558
User-defined display function 141, 152
User-defined types 312, 339
USES statement 6

V

Validation routine 313, 344
VDISK.SYS 588
VERIFY 512
VGA 30, 47, 49, 72, 75
Video attributes 33, 84
Video memory 31, 57
Video mode 24, 30, 50, 67
Video pages 19, 21, 30, 75, 215, 229, 260, 265, 282
VideoSegment 21
Virtual screens 20, 89, 260, 264, 266, 749
Visible display page 260, 282

W

Warranty period 18
White space 461, 474
Wilton, Richard 7
Window coordinates 28, 281, 284
Window, popup and tiled 83, 100
Window stack 83
Windows, Microsoft 29
WINEXMPL 749
WINWOW 749
Word delimiters 463
Word wrap 402, 464
WordStar 153, 165
Wrap mode 301, 375

Z

Zenith 21
Zero-based arrays 630

TURBO PASCAL

A Powerful Library for Turbo Pascal

Turbo Professional is a library of over 600 essential procedures and functions, optimized specifically for non-object-oriented Turbo Pascal programming. You get over 50,000 lines of perfected code including:

- Pop-up resident routine
- Windows and menus
- Data entry routines
- BCD arithmetic
- Interrupt service routines
- Context sensitive help
- EMS and extended memory access
- Long strings
- Large arrays
- Macros
- Fast sorting
- Much, much more

Powerful Programs, Faster and Easier

With these routines, you'll develop more powerful programs, and save time and effort. You can concentrate on the creative aspects of the job and let Turbo Professional handle the low level chores.

Hot Demo Programs, Too

Turbo Professional also includes more than ten commercial quality programs that illustrate how to use the routines in your programs. You'll find a wealth of specific examples in:

- A memory-resident keyboard macro processor
- A memory-resident programmer's quick reference chart
- A memory-resident programmer's calculator
- A memory-resident programmer's time and billing program
- An interactive menu maker that writes source code for you
- A bullet proof full-screen data entry demo with pop up help

Documentation: Clear and Complete

Turbo Professional includes clear, comprehensive documentation and pop-up help. The calling syntax, dependencies, and operation of every routine are described. You get free expert telephone and CompuServe support. Full source code is included and you pay no royalties when you use these routines.

Turbo Professional requires Turbo Pascal 4.0, 5.0, 5.5, 6.0, 7.0 or Borland Pascal 7.0. MS-DOS 2.0 or later and an IBM, PC, XT, AT, PS/2 or compatible are required.

TURBO Power

TurboPower Software
P.O. Box 49009
Colorado Springs, CO 80949-9009
CompuServe I.D. 76004, 2611

© TurboPowerSoftware 1993